А. Ф. Верлань, И. А. Чмырь
С. Д. Кузниченко, Л. Б. Коваленко

# ИМПЕРАТИВНОЕ ПРОГРАММИРОВАНИЕ И ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ МОДЕЛИРОВАНИЕ: JAVA, UML, OCL

Одесса
2014

УДК 681.3.
ББК 32.973
В33

Рекомендовано Министерством образования и науки Украины *как учебное* пособие для высших учебных заведений *(письмо № 1/11-11840 от 22.07.13)*

Рецензенты: д. т. н., проф. ***Томашевский В. Н.;***
д. т. н., проф. ***Нестеренко С. А.;***
к. т. н., доц. ***Волощук Л. А.***

**Верлань А. Ф., Чмырь И. А., Кузниченко С. Д., Коваленко Л. Б.**

В33 Императивное программирование и объектно-ориентированное моделирование : Java, UML, OCL : Учебное пособие для студентов высших учебных заведений. — Одесса : Издательство, 2014. 232 с.

ISBN

В учебном пособии изложены вопросы моделирования и кодирования методов и классов на языке программирования Java, а также теоретические основы и практические примеры объектно-ориентированного моделирования в среде унифицированного языка моделирования UML (Unified Modeling Language) и языка объектных ограничений OCL (Object Constraint Language). Отличительной особенностью пособия является обучение программированию по принципу «от модели к коду». Пособие ориентировано на читателя, не имеющего предварительных знаний и навыков в области программирования, и рассчитано на студентов, обучающихся по направлениям «Компьютерные науки» и «Компьютерная инженерия». Пособие может использоваться в высших и средних учебных заведениях при преподавании дисциплин, посвященных введению в программирование и алгоритмизацию, а также основам объектно-ориентированного программирования.

**УДК 681.3.**
**ББК 32.973**



ISBN

# ОГЛАВЛЕНИЕ

# Предисловие

Настоящее учебное пособие рассматривается авторами как введение в программирование на основе языка программирования Java, а не как введение в язык программирования Java. Пособие состоит из двух частей. В первой части изучаются лексические и синтаксические основы языка Java, а также основы моделирования и кодирования ряда общеупотребительных алгоритмов, а во-второй — принципы объектно-ориентированной разработки программ и основы моделирования структур объектных программ.

Тот вид программирования, который изучается в первой части пособия, соответствует императивной парадигме программирования. Целью изучения императивного программирования, как оно рассматривается в первой части пособия, является получение знаний и навыков, достаточных для разработки небольших программ-методов, которые, в рамках объектно-ориентированной парадигмы, являются основными функциональными элементами программных систем. Целью второй части пособия является изучение основных идей объектно-ориентированной парадигмы программирования, а также средств моделирования объектно-ориентированных систем. Знания и навыки, полученные при изучении материала второй части, являются необходимой основой для последующего изучения вопросов кодирования объектно-ориентированных программ.

Изложение материала учебного пособия строится, по преимуществу, «от модели — к коду» и, по возможности, «от общего к частному». При таком подходе к изучению программирования главные творческие усилия по решению задач программирования сосредотачиваются на этапе синтеза модели, а не на этапе написания программного кода. Когда модель синтезирована, то задача, по сути, уже решена. Код необходим только для того, чтобы поручить компьютеру реализовать решение. Чем точнее и формализованнее представлена модель, тем проще отобразить ее в программный код.

Для разработки моделей в пособии используются средства моделирования, представляющие собой комбинацию унифицированного языка моделирования UML (Unified Modeling Language) и языка объектных ограничений OCL (Object Constraint Language). Отмеченные языки являются сегодня фактическим стандартом разработки коммерческих программных проектов. В пособии, по мере необходимости, изучаются те элементы UML и OCL, которые необходимы для понимания последующего материала.

Унифицированный язык моделирования UML является диаграмматическим, и модели, специфицированные в этом языке, представляют собой диаграммы, построенные с использованием графических символов и правил, принятых в UML.

Диаграмматическое представление модели наглядно, легко и быстро воспринимается человеком и в концентрированном виде содержит большое количество информации относительно моделируемой системы. Однако, как правило, одних UML-диаграмм недостаточно для их корректного отображения в завершенный программный код. Язык объектных ограничений OCL является символьным языком и разработан как дополнение к унифицированному языку моделирования UML, позволяющее уточнять диаграмматические модели при помощи предложений, формулирующих ограничения, в рамках которых должны функционировать те или иные модельные элементы. Комбинируя UML-диаграммы, специфицирующие структуру, и OCL-предложения, специфицирующие ограничения, можно получить модель, содержащую минимум неопределенности и легко отображаемую в программный код.

В первой части пособия язык моделирования UML используется для моделирования алгоритмов программ-методов. Модели алгоритмов представляются в виде UML-диаграмм деятельности. Модели алгоритмов конструируются из небольшого количества базовых поведенческих структур на основе идей структурного программирования. Во второй части пособия язык моделирования UML используется для моделирования пространственной структуры класс-структурированной программной системы. Структура программной системы представляется в виде UML-диаграммы классов и уточняется OCL-ограничениями.

Пособие не следует рассматривать как справочник. Оно построено таким образом, что предполагает последовательное и систематическое изучение материала. Однако, после первого прочтения пособие может использоваться как справочник. Чтение книги не предполагает знакомства с программированием, кроме необходимого школьного минимума в рамках курса Информатики. Поэтому пособие может использоваться при изучении программирования на младших курсах высших учебных заведений. Авторы используют материал первой части пособия при преподавании дисциплины «Основы алгоритмизации и программирования» в двухсеместровом курсе первого года обучения, а вторую — в односеместровом курсе второго года обучения при преподавании дисциплины «Объектно-ориентированное программирование. Часть 1».

Каждый раздел пособия завершается списком упражнений для самостоятельной работы. Эти упражнения могут использоваться как для самостоятельной работы студентов, так и для аудиторной работы с преподавателем. Упражнения, которые размещены после разделов первой части (кроме упражнений разделов 1 и 2), предполагают кодирование и могут использоваться преподавателями для организации лабораторных работ. В этом случае студенты должны быть снабжены шаблоном кода, внутри которого они должны писать исходный текст программы, редактировать, компилировать и выполнять программный код. Упражнения, которые размещены после разделов второй части пособия, предполагают решение упражнений на бумага или доске и могут использоваться преподавателями для организации практических занятий.

Учебное пособие является самодостаточным, однако для углубленного изучения материала в конце пособия приведен список дополнительных источников.

ЧАСТЬ 1

# ИМПЕРАТИВНОЕ ПРОГРАММИРОВАНИЕ

Раздел 1

# ВВЕДЕНИЕ

Первая часть посвящена императивному программированию на языке Java. Целью императивного программирования является решение задачи разработки относительно небольших программ-методов, которые могут иметь самостоятельную ценность, но которые, главным образом, предназначены для реализации поведения больших объектно-ориентированных программных систем. Вторая часть пособия посвящена изучению основ объектно-ориентированного моделирования, лежащего в основе объектно-ориентированной технологии программирования. Объектно-ориентированное программирование нацелено на разработку и кодирование больших программных систем и является сегодня доминирующей парадигмой программирования. Однако, объектно-ориентированные идеи настолько глубоко проникли в структуру языка программирования Java, что изучение императивного программирования на этом языке очень сложно, если вообще возможно, без знакомства с основами объектно-ориентированной парадигмы программирования.

## 1.1. Парадигмы программирования

Компьютерную программу, в общем случае, можно мыслить как композицию данных и алгоритмов. Данные используются для представления порций информации, а алгоритмы связывают между собой последовательности действий над данными в цепи, которые реализуют функции программы. В процессе эволюции дисциплины программирования были предложены различные принципы взаимной организации данных и алгоритмов. Эти принципы получили наименование парадигмы программирования. Наиболее известны следующие парадигмы: императивная, процедурная, декларативная, логическая, функциональная и объектно-ориентированная. Объектно-ориентированная парадигма сегодня доминирует при разработке коммерческих программных систем. На основе объектно-ориентированной парадигмы развиваются такие парадигмы, как компонентное программирование, событийное программирование и др.

*Императивная парадигма* является исторически первой парадигмой программирования. Первые поколения программ на алгоритмических языках высокого

уровня разрабатывались в рамках императивной парадигмы. Иногда императивную парадигму программирования называют также *алгоритмической парадигмой программирования*. Императивная парадигма предполагает, что программист может использовать небольшое количество данных различных типов, которые встроены в язык программирования. Такие данные называются *примитивными данными*. Императивная парадигма не позволяет программисту вводить в язык программирования новые типы данных. В программном коде примитивные данные представлены переменными и константами. Предполагается, что местом хранения примитивных данных является основная память компьютера. Алгоритмы программы составлены из предложений, включающих *операторы*, предназначенные для выполнения арифметических, логических и других операций над переменными и константами. Из переменных, констант и операторов конструируются *выражения*, подобные выражениям, используемым в арифметике или алгебре. Выражение имеет возвращаемое значение. *Возвращаемое значение выражения* — это результат выполнения действий, кодируемых операторами в выражении. Одним из важнейших операторов императивного программирования является *оператор присваивания*, при помощи которого в переменные записываются новые значения, например, возвращаемые значения выражений. Кроме оператора присваивания, а также операторов арифметических и логических операций, императивная парадигма программирования предполагает некоторое количество операторов, управляющих последовательностью выполнения предложений в алгоритме. Сюда входит оператор с именем «`if`», при помощи которого кодируются альтернативные последовательности предложений, и оператор с именем «`while-do`», при помощи которого кодируются циклы или многократно выполняемые участки кода. Императивная парадигма программирования предполагает, что весь алгоритм разбивается на последовательность логически завершенных шагов, и программист, используя операторы и данные, должен кодировать каждый шаг алгоритма.

Пусть, например, необходимо написать программу, вычисляющую значение гипотенузы прямоугольного треугольника, если известны значения его катетов. В рамках императивной парадигмы программирования необходимо вначале разработать алгоритм этих вычислений, а затем отобразить этот алгоритм в программный код, используя необходимые данные и операторы. Алгоритм вычисления гипотенузы будет включать алгоритм приближенного вычисления квадратного корня. Для того, чтобы разработать такой алгоритм, программисту, наверное, придется вспомнить, каким способом можно вычислить приближенное значение квадратного корня.

*Процедурная парадигма* может рассматриваться как дальнейшая эволюция императивной парадигмы. Все, что было сказано ранее о данных и алгоритмах относительно императивной парадигмы, справедливо и по отношению к процедурной парадигме. Однако, имеется существенное отличие. Процедурная парадигма позволяет программисту расширять операционную часть языка программирования, включая в него свои собственные процедуры. Таким образом, процедурное программирование предполагает, что при отображении алгоритмов в программный код

программист может использовать не только операторы, встроенные в язык программирования (арифметические, присваивание, `if` и др.), но и более крупные операционные конструкции — программные процедуры. Программная процедура — это отдельная программа, реализующая некоторый логически завершенный алгоритм. Иногда термин *программная процедура* является синонимом термина *программная функция*. Процедуры, обычно, реализуют те алгоритмы, которые часто используются при кодировании. Так, например, если задача заключается в разработке кода, вычисляющего значение гипотенузы при известных катетах, то программист может заранее написать процедуру, вычисляющую значение квадратного корня, присвоить ей имя и в том месте кода, где необходимо вычислить квадратный корень, вызвать эту процедуру, используя ее имя. После вызова процедуры выполнение предложений основного кода приостанавливается и начинают выполняться предложения процедуры. После завершения работы процедуры продолжается выполнение предложений основного кода.

Процедурная парадигма программирования предполагает широкое использование заранее подготовленных процедур. Поэтому развитие этой парадигмы приводит к накапливанию процедур в *библиотеках процедур*. Идея библиотек процедур очень практична, поскольку позволяет программисту использовать не только свои собственные, но и *предопределенные* (созданные другими) библиотеки процедур.

Развитие процедурной парадигмы привело к появлению эффективной технологии разработки как отдельных процедур, так и всей программы, называемой *структурное программирование*. Структурное программирование основано на утверждении о том, что структура любого алгоритма может быть составлена из небольшого количества типовых алгоритмических структур. Выяснилось, что достаточно только трех *основных алгоритмических структур*, чтобы представить любой, сколь угодно сложный алгоритм: (1) структура типа блок; (2) структура типа развилка и (3) структура типа цикл, с проверкой вначале. Алгоритмические структуры изображаются графически, что позволяет использовать диаграмматическое представление алгоритмов. Алгоритмические языки программирования, которые были разработаны после появления технологии структурного программирования, учитывают основные идеи этой технологии. К таким языкам относится, например, язык программирования Pascal. Операторы языка Pascal позволяют легко отображать графическое представление основных алгоритмических структур в программный код. Технология структурного программирования придает алгоритмической части программы ряд достоинств. Алгоритмы имеют понятную логическую организацию и легко воспринимаются не только их авторами, но и сторонними пользователями. Отображение таких алгоритмов в код осуществляется весьма просто и может быть автоматизировано. Необходимо только заменить графические изображения алгоритмических структур соответствующими предложениями кода.

*Декларативная парадигма* основана на философии, которая диаметрально противоположна философии императивного и процедурного программирования.

Программирование в рамках императивной и/или процедурной парадигмы исходит из предположения, что компьютер не в состоянии самостоятельно решить

проблему, а может только реализовывать решение, найденное программистом. Программист решает проблему и представляет решение в виде кода. Когда написан код, то проблема, по сути, уже решена. Необходимо только «материализовать» решение. И эта работа поручается компьютеру. Задача компьютера — точно и быстро выполнить действия, предусмотренные кодом.

Декларативная парадигма исходит из предположения, кто компьютер может сам определять последовательность действий, необходимых для решения проблемы, если предоставить ему подробное описание проблемы. Задача программиста радикально меняется. Его внимание сосредотачивается на разработке подробного описания того, что он хочет получить от компьютера. Поэтому, в отличие от императивной парадигмы, в декларативной парадигме входные данные являются неизменными, а алгоритмы подвергаются изменениям. В рамках императивной парадигмы изменению подвергаются входные данные, а алгоритмы остаются неизменными. Декларативная парадигма предполагает, что входные данные-описатели передаются программе и остаются неизменными при ее выполнении. Вместо того, чтобы модифицировать входные данные, программа определяет действия с целью продуцирования результатных данных, которые специфицированы во входных данных-описателях.

Декларативная парадигма воплощена в языке структурированных запросов SQL (Structured Query Language). Используя SQL, можно подробно специфицировать данные, которые необходимо получить из базы данных, а система управления базой данных сама формирует последовательность действий, которые нужно выполнить, чтобы, во-первых, извлечь требуемые данные из базы и, во-вторых, сформировать их в том виде, который декларативно описан в SQL-запросе. Во второй части книги изучается язык объектных ограничений OCL (Object Constraint Language), который является декларативным языком и предназначен для специфицирования ограничений, которым должна удовлетворять программа в процессе выполнения. Логическую и функциональную парадигмы программирования можно рассматривать как дальнейшее развитие декларативной парадигмы.

*Логическая парадигма* предполагает, что программирование заключается в специфицировании проблемы и ожидаемого решения проблемы, а программа сама синтезирует алгоритм, необходимый для ее решения и получения результатных данных. Логическая парадигма предполагает также, что универсальным алгоритмом решения проблем является алгоритм логического вывода дедуктивного типа.

В рамках логической парадигмы специфицирование проблемы осуществляется при помощи набора предложений двух видов. Первый вид предложений представляет собой набор известных фактов относительно проблемной области, а второй вид предложений — это правила вывода, связывающие факты между собой. Ожидаемое решение проблемы специфицируется при помощи предложения-запроса. Набор предложений-фактов и предложений-правил вывода рассматриваются как аксиомы. Предложения-запросы трактуются как теоремы, подлежащие доказательству. Работа программы заключается в доказательстве сформулированной теоремы в рамках заданного набора фактов и правил вывода.

Ценность и значимость логической парадигмы программирования заключается в том, что она является общей платформой, объединяющей такие дисциплины как искусственный интеллект, математическую логику и программирование. Наборы фактов и правил вывода, связывающих эти факты, являются универсальным средством представления знаний которое, в искусственном интеллекте часто используется для создания баз знаний. Технология логического программирования позволяет кодировать системы искусственного интеллекта, работа которых основана на взаимодействии с базой знаний при помощи серии транзакций. Отдельная транзакция состоит из запроса к базе знаний и ответа, являющегося результатом логического вывода из тех фактов и правил, которые хранятся в базе знаний. Наиболее известным языком логического программирования является Prolog.

*Функциональная парадигма* программирования базируется на понятии *математической функции*, а работа программы трактуется как вычисление значений математических функций. Математические функции, используемые в функциональном программировании, возвращают значения, которые зависят только от входных параметров. Функциональная парадигма предполагает, что компьютерная программа может решать проблемы только путем вычисления возвращаемых значений функций, у которых входными параметрами являются возвращаемые значения других функций. Функциональная парадигма не предполагает явного хранения состояния программы, поэтому языки функционального программирования не используют ни переменных, ни оператора присваивания.

В функциональном программировании функция рассматривается как алгоритм, вычисляющий неизвестное значение функции из известных входных параметров, которая не обладает *побочным эффектом*. В императивном и процедурном программировании побочным эффектом от работы программной функции называется *изменение состояния программы* после завершения работы функции. Эффект называется побочным, поскольку основным эффектом от работы программной функции является формирование того значения, которое функция возвращает после завершения своей работы. Побочный эффект возникает, поскольку императивная парадигма разрешает программной функции использовать не только переданные ей входные параметры, но и значения переменных программы.

Отличительной особенностью функциональной парадигмы программирования является отсутствие средств организации циклов для кодирования многократно выполняемых действий. В тех случаях, когда необходимо обеспечить многократное выполнение некоторой функции, используются *рекурсивные функции*. Рекурсивные функции вызывают сами себя, позволяя операции выполняться снова и снова.

Функциональное программирование, так же как и логическое программирование, редко используется для разработки коммерческих программ. Поле деятельности этой парадигмы программирования — разработка приложений в области искусственного интеллекта и моделирование компьютерных систем. Наиболее известным языком функционального программирования является язык LISP (LISt Processing language).

### 1.1.1. Объектно-ориентированная парадигма в языке Java

Объектно-ориентированная парадигма программирования предполагает, что программа состоит из *программных объектов*. Программные объекты размещаются в основной памяти компьютера и взаимодействуют между собой. Программный объект мыслится как *капсула*, содержащая данные и алгоритмы. Принято данные, входящие в состав объекта, называть *полями объекта*, а алгоритмы — *методами объекта*. Используя объектно-ориентированную терминологию, можно сказать, что *объект инкапсулирует поля и методы*. Поля и методы включаются в объект не случайным образом. Они логически связаны и позволяют использовать объект для моделирования различных сущностей окружающего мира и использовать их при конструировании компьютерной программы. Значения полей объекта определяют *состояние объекта*, а методы — *поведение объекта*. Например, такую геометрическую фигуру как прямоугольный треугольник можно моделировать при помощи программного объекта. Структура простого варианта такого объекта может включить два поля: (1) поле, хранящее значение первого катета, и (2) поле, хранящее значение второго катета, и два метода: (1) метод, вычисляющий гипотенузу, и (2) метод, вычисляющий площадь. Состояние рассматриваемого объекта определяется конкретными значениями его катетов, а поведение — умением вычислять гипотенузу и площадь.

Объекты взаимодействуют друг с другом посредством обмена *сообщениями*. Объекты обмениваются сообщениями не случайным образом, а целенаправленно. Целью обмена сообщениями является реализация функций всей программной системы, включающей множество объектов. Реакцией объекта, получившего сообщение, является выполнение одного из своих методов. Сообщения, которыми обмениваются объекты, адресованы методам объектов и являются, по сути, средством вызова того или иного метода.

Несмотря на то, что рассматриваемая парадигма программирования носит наименование объектно-ориентированной, программист, занятый разработкой объектно-ориентированной программы, не кодирует отдельные объекты, необходимые для реализации функций программы. Главным предметом внимания программиста является конструирование *классов объектов*. Под классом объектов будем понимать множество подобных объектов, которые имеют одинаковые наборы полей (но с различными значениями) и одинаковые наборы методов. Если класс объявлен, то, используя его как шаблон, можно легко создать сколь угодно много структурно-подобных объектов. Поэтому, в ряде случаев, вместо термина объект используется термин *экземпляр класса*. Класс является абстракцией. К понятию класс не применимо понятие состояние. Класс не выполняет действия, реализующие функции программы. Он нужен, главным образом, для создания объектов. Существует бесконечно много прямоугольных треугольников, отличающихся значениями своих катетов, но все они могут быть описаны при помощи одного класса.

Как было отмечено выше, совокупность значений полей объекта есть его состояние. Объект, созданный при помощи класса, получает некоторые исходные значения полей, определяющие его *начальное состояние*. В процессе «жизнедеятельности» объекта его состояние изменяется. Поля и методы объекта инкапсулированы. С точки зрения состояния объекта, инкапсуляция означает, что объект не позволяет другим объектам-клиентам получать непосредственный доступ к своим полям и, таким образом, бесконтрольно изменять его состояние. Изменить состояние объекта можно только опосредованно при помощи методов этого объекта. В этом случае объект может контролировать изменение своего состояния. Часто в состав объекта вводятся специальные методы, предназначенные только для обеспечения опосредованного доступа к его полям.

Проиллюстрируем некоторые понятия объектно-ориентированной парадигмы на примере объявления класса на языке программирования Java. Предположим, что программа включает множество взаимодействующих объектов, моделирующих группу личностей. В контексте объектно-ориентированной парадигмы это означает, во-первых, что в программе должен быть объявлен класс личностей, а во-вторых, что в процессе работы программы необходимо создавать экземпляры этого класса в зависимости от потребностей программы. На рис. 1.1 приведено объявление простого класса личностей. Код, приведенный на рис. 1.1, не предназначен для детального изучения и может быть непонятен в деталях на начальном этапе изучения программирования. Его задача — проиллюстрировать понятийный аппарат объектно-ориентированной парадигмы на конкретном примере Java-кода.

Объявление класса начинается с заголовка, в котором после служебного слова `class` указывается имя класса. Класс, объявленный на рис. 1.1, имеет имя `Person` (личность). В заголовке также указываются общие атрибуты и характеристики класса и его отношения с другими классами программы. Поэтому заголовок класса может быть значительно длиннее, чем тот, который приведен на рис. 1.1.

После заголовка класса на рис. 1.1 записаны предложения, при помощи которых объявлены следующие структурные элементы класса:

- поля объектов класса;
- поле класса;
- конструктор объектов;
- методы объектов класса.

Перечисленные структурные элементы класса объединены в блок, начало которого отмечено левой фигурной скобкой, а конец — правой фигурной скобкой.

В классе могут быть объявлены два вида полей: поля объектов и поля класса. *Поля объектов* характеризуют любой и каждый объект класса как отдельную самостоятельную сущность. В классе на рис. 1.1 объявлены два поля объектов с именами `name` (имя) и `gender` (пол) и, следовательно, каждый объект класса `Person` будет содержать по два поля с конкретными значениями. Например, для некоторого объекта эти поля могут иметь значения: «Владимир Шарапов» и «мужчина», соответственно, а для другого — «Софья Ковалевская» и «женщина».

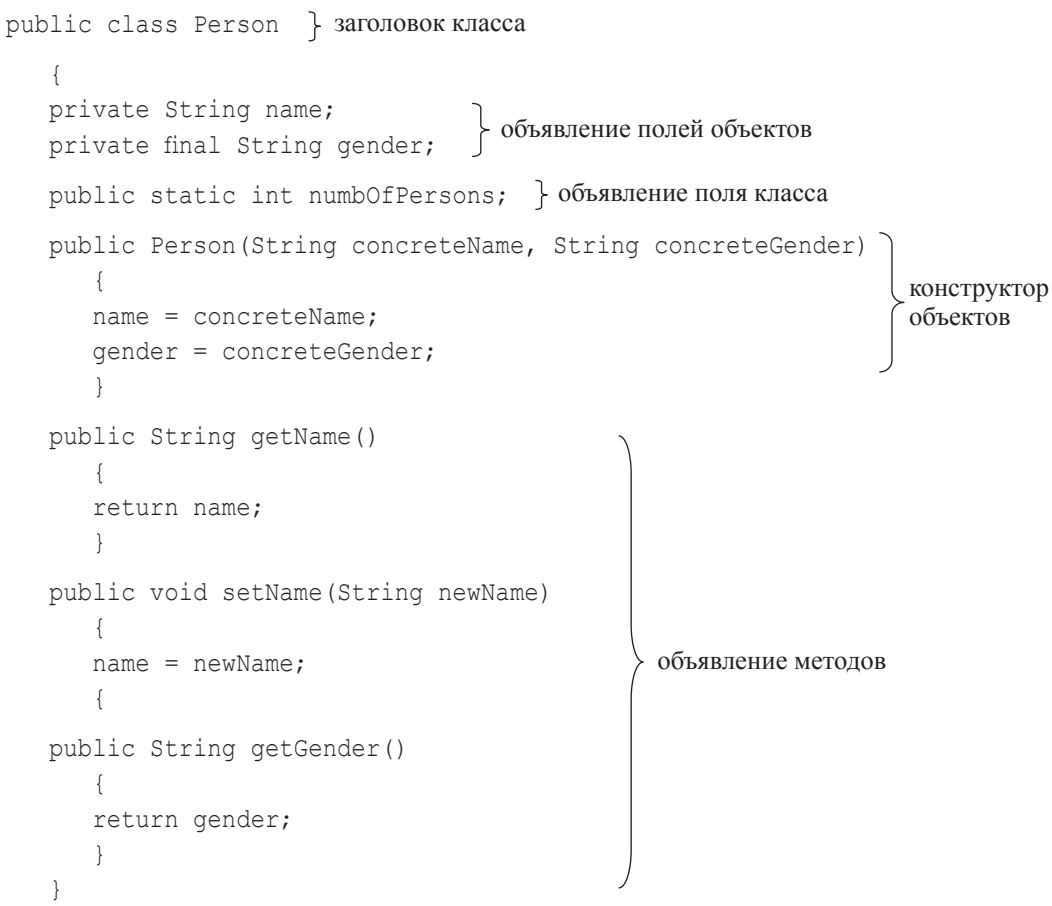



**Рис. 1.1.** Объявление класса на языке программирования Java

*Поля класса*, называемые также *статическими полями*, характеризуют весь класс объектов как отдельную сущность. В классе `Person` объявлено одно статическое поле с именем `numbOfPersons` (количество личностей), значением которого является целое положительное число, равное количеству объектов, созданных при помощи класса `Person`. Ясно, что вне зависимости от количества экземпляров объектов класса он содержит только один экземпляр статического поля.

Важным структурным элементом класса является особый метод, называемый *конструктор объектов*. Конструктор отличается от всех остальных методов тем, что его имя всегда совпадает с именем класса. Конструктор объектов участвует в процессе создании объектов класса. Его главная задача — записать в поля объектов их начальные значения. Этот процесс называется процессом *начальной инициализации полей*. В «жизни» объекта метод-конструктор участвует только один раз при его «рождении».

Конструктор вызывается в специальном предложении со служебным словом `new`. Такое предложение должно включать: (1) имя создаваемого объекта; (2)

имя класса, который используется для создания объекта и (3) вызов конструктора. Приведенный ниже фрагмент кода иллюстрируют использование предложения со служебным словом `new` для создания двух объектов класса `Person`.

```
Person vladimir = new Person("Владимир Шарапов", "мужчина");
Person sofia = new Person("Софья Ковалевская", "женщина");
```

Первое предложение кодирует создание объекта, который будет иметь имя `vladimir`. Имя нового объекта указано в левой части предложения. В правой части, после служебного слова `new`, записан вызов конструктора, которому передаются начальные значения полей `name` и `gender`.

Второе предложение кодирует создание объекта, который будет иметь имя `sofia`. При вызове конструктора ему передаются начальные значения полей `name` и `gender` для объекта `sofia`.

Класс может включать несколько конструкторов, отличающихся тем, каким образом они осуществляют начальную инициализацию. Объявление конструктора, как и объявление остальных методов, начинается с заголовка, содержащего имя конструктора, после которого следует блок с предложениями, кодирующими действия конструктора. Начало и конец блока обозначаются фигурными скобками.

Список методов класса `Person`, объявленных на рис. 1.1, состоит только из специальных методов, обеспечивающих опосредованный доступ к полям объектов этого класса. Эти методы называются *стандартными*, и их имена, как правило, начинаются со слов `get` (получить) и `set` (установить). Например, внешний объект-клиент для того, чтобы прочитать значение поля `name`, должен использовать метод `getName` этого объекта, а для того, чтобы записать в это поле новое значение, — метод `setName`.

После того, как созданы объекты, к ним могут направляться сообщения, вызывающие методы этих объектов. Сообщение кодируется при помощи выражения, состоящего из имени объекта и имени вызываемого метода, между которыми записывается разделительный символ в виде точки. Например, после того, как созданы объекты `vladimir` и `sofia`, к ним можно направить следующие сообщения.

```
vladimir.getName()
sofia.getName()
sofia.setName("Софья Лобачевская")
```

Первое сообщение вызывает метод `getName` в объекте `vladimir`, который возвращает значение поля `name` этого объекта, а именно строку: «Владимир Шарапов».

Второе сообщение вызывает метод `getName` в объекте `sofia`, который возвращает значение поля `name` объекта `sofia` в виде строки: «Софья Ковалевская».

Третье сообщение вызывает метод `setName` в объекте `sofia`. Методу `setName` передается новое значение поля `name`, которое должно заменить предыдущее значение. Этот метод не возвращает никаких значений, а записывает в поле `name` новое значение: «Софья Лобачевская».

Стандартные методы определяют стандартное поведение объекта, заключающееся в том, что они, при условии инкапсуляции объекта, обеспечивают возможность чтения значений полей объекта и записи в поля новых значений. Кроме стандартных методов, классы, обычно, включают объявления нестандартных методов, определяющих специфическое поведение объектов этого класса. В примере класса `Person` на рис. 1.1. нестандартные методы не объявлены.

При разработке объекно-ориентированной программы важными является ответ на вопрос о том, какие классы необходимо объявить, чтобы их объекты были в состоянии реализовать все функции программы, а также ответ на вопрос о том, какую структуру должен иметь каждый из классов. В состав системы программирования языка программирования Java включено большое количество *предопределенных классов*, которые содержат объявления полей и методов, необходимых для реализации функций программы. Таким образом, при разработке объекно-ориентированной программы из предопределенных классов заимствуются уже готовые элементы. Чем большими знаниями и навыками в области объекно-ориентированного программирования обладает программист, тем большая часть его программы построена с использованием предопределенных классов. Например, нет необходимости разрабатывать метод, выделяющую подстроку из заданной строки, или метод, вычисляющий квадратный корень вещественного числа. Эти методы можно заимствовать из предопределенных классов `String` и `Math` соответственно.

Современные языки программирования, используемые при разработке коммерческих программ, трудно отнести к какой-либо одной из перечисленных парадигм. Как правило, они являются универсальными по отношению к парадигмам программирования и включают средства, позволяющие кодировать программы, придерживаясь различных парадигм. Например, язык программирования Java хоть и называется объектно-ориентированным, однако кодирование с использованием языка Java не гарантирует автоматического получения объектно-ориентированной программы. Объектно-ориентированным является не язык программирования, а парадигма программирования. Поэтому на языке программирования Java можно более или менее успешно кодировать программы в рамках любой из перечисленных парадигм.

## 1.2. Императивное программирование на языке Java

Изучение программирования на основе языка Java целесообразно начинать с тех средств, которые предназначены для кодирования программ в рамках императивной парадигмы. Знания и навыки в области императивного программирования позволяют, во-первых, познакомиться со значительной частью синтаксических конструкций языка и, во-вторых, научиться кодировать небольшие программы-методы которые, на последующих этапах изучения Java будут использоваться как главные

структурные элементы больших объектно-ориентированных программных систем. При кодировании классов наиболее трудоемкой частью является объявление методов. Поэтому освоение навыков императивного программирования является необходимым первым шагом, который значительно облегчает изучение объектно-ориентированного программирования.

Трудность изучения императивного программирования на языке Java заключается в том, что в императивном программировании на Java присутствуют элементы и процедурного, и объектно-ориентированного программирования. При кодировании программ-методов в рамках императивной парадигмы программист решает двоякую задачу. Во-первых, необходимо синтезировать тот набор данных, который достаточен для представления любого состояния программы, а во-вторых, необходимо синтезировать алгоритм, который в процессе манипулирования этими данными, реализует функции программы. Императивная парадигма предполагает, что для этой цели программист должен использовать только данные и операторы, встроенные в язык программирования. Однако, в языке программирования Java некоторые важные данные не встроены в язык программирования, а являются объектами предопределенных классов, а некоторые важные операции можно выполнить только путем вызова методов предопределенных классов.

Например, такой важный тип данных как строковые данные, при помощи которых в программе представляются строки символов, объявлен как предопределенный класс с именем `String`. Поэтому для того, чтобы использовать в программе данные типа `String`, программист должен уметь создавать объекты этого класса, а для того, чтобы манипулировать данными типа `String`, программист должен знать, какие методы объявлены в классе `String`, а также как их вызывать и как, с их помощью, манипулировать строками. Отмеченные знания и умения относятся к области процедурной и объектно-ориентированной парадигм.

Другим примером использования процедурной и объектно-ориентированной парадигмы является случай применения массивов данных при кодировании методов. Массивом называется набор однотипных и индексированных данных. Индекс — это уникальный номер данного в наборе, который обеспечивает удобный доступ к элементам массива. Поэтому массивы часто используются в практике императивного программирования. В языке программирования Java массивы являются объектами предопределенного класса массивов. Поэтому для использования массивов, при кодировании методов необходимо обладать некоторыми знаниями в области объектно-ориентированного программирования.

При изучении императивного программирования и при кодировании методов в качестве устройства ввода данных в программу и вывода данных из программы обычно используются клавиатура и экран консоли компьютера. Консолью компьютера называются те периферийные устройства, при помощи которых пользователь компьютера может взаимодействовать с его операционной системой. В современных компьютерах консоль включает клавиатуру и экран монитора. В языке программирования Java клавиатура и экран консоли моделируются объектами предопределенных классов. Например, для кодирования вывода на экран консоли часто

используются методы предопределенного класса `PrintStream`, а для кодирования ввода с клавиатуры консоли — методы предопределенного класса `Scanner`.

Как было отмечено ранее, в рамках процедурной парадигмы программирования была предложена эффективная технология синтеза алгоритмов программ, названная структурным программированием. Структурное программирование как общая технология разработки больших программных систем уступило место объектно-ориентированной технологии, однако структурное программирование остается эффективным способом разработки и кодирования методов. Напомним, что структурное программирование основано на утверждении о том, что структура любого алгоритма может быть составлена из трех типовых алгоритмических структур: (1) структура типа блок; (2) структура типа развилка и (3) структура типа цикл, с проверкой вначале. Язык программирования Java содержит средства, позволяющие расширить этот список. В последующих разделах первой части книги мы вернемся к структурному программированию и изучим варианты структуры типа развилка и структуры типа цикл. Дополнительные варианты алгоритмических структур позволяют, в некоторых случаях, синтезировать алгоритмы, отображаемые в более компактный код. Успешное использование технологии структурного программирования предполагает наличие средств для графического представления как алгоритмических структур, так и алгоритмов, составленных из этих структур. Для этой цели мы будем использовать средства унифицированного языка моделирования UML (Unified Modeling Language). Язык UML является диаграмматическим и позволяет моделировать различные элементы и аспекты программных систем при помощи серии диаграмм. Одна из диаграмм UML, называемая «диаграмма деятельности», является удобным средством для графического представления алгоритмических структур и синтеза алгоритмов на их основе. Следуя терминологии UML, далее в тексте вместо словосочетания «основные алгоритмические структуры» будем использовать словосочетание «основные поведенческие структуры».

Предметом императивного программирования на языке Java является программа-метод. Термин метод используется, главным образом, в рамках объектно-ориентированной парадигмы и является синонимом терминов программная функция и программная процедура, принятых в рамках процедурной парадигмы. Метод — это небольшая, логически завершенная программа, решающая конкретную задачу. Например, целесообразно вычисление значений элементарных математических функций представить в виде набора методов.

В общем случае, при вызове метода ему передаются значения входных параметров, а после завершения работы метода он возвращает найденное значение. В языке программирования Java принято, что *метод может иметь несколько входных параметров, но только одно возвращаемое значение*. В частном случае у метода могут отсутствовать входные параметры и он может не иметь возвращаемого значения. В языке программирования Java методы не существуют самостоятельно и независимо, а объявляются внутри класса.

На рис. 1.2 приведена структура кода метода для случая, когда метод возвращает значение. Код метода состоит из заголовка метода и тела метода. Если метод

возвращает значение, то в его заголовке специфицируются: имя метода, список входных параметров, тип возвращаемого значения и общие свойства метода, описываемые модификаторами метода. В правой части заголовка располагается *имя метода*, за которым следует *список входных параметров* в круглых скобках. Элементы списка разделяются символом «запятая». Каждый элемент списка входных параметров состоит из имени типа параметра и имени параметра. Если метод не имеет входных параметров, то скобки остаются пустыми. Принято параметры, используемые при объявлении метода, называть *формальными параметрами*, а параметры, используемые при вызове метода, — *фактическими параметрами*. Имя метода и типы входных параметров метода называются *сигнатурой метода*. Сигнатура метода уникально идентифицирует метод. При вызове метода используется не имя метода, а его сигнатура.

```
<модификаторы><тип возвращ. значен.><имя метода>(<входн. параметры>)
   {
   <предложения тела метода>
   return <величина возвращаемого значения>;
   }
```

**Рис. 1.2.** Структура кода метода с возвращаемым значением

Перед именем метода располагается *тип возвращаемого значения*. Следует обратить внимание на то, что в заголовке метода указывается не имя возвращаемого значения, а его тип. Если метод возвращает некоторое значение, то оно всегда обладает типом, но может не иметь имени. Например, в теле метода может быть вычислено значение арифметического выражения, возвращаемое значение которого и будет возвращаемым значением метода, но в этом случае возвращаемое значение может не иметь имени.

Если метод возвращает некоторое значение, то оно должно быть специфицировано в последнем предложении метода при помощи оператора `return`. Оператор `return` прерывает выполнение последовательности предложений и завершает работу метода. Величина возвращаемого значения указывается в операторе `return` и, в общем случае, специфицируется в виде выражения. В частном случае величина возвращаемого значения может быть специфицирована при помощи имени переменной или литера. Ясно, что тип возвращаемого значения выражения должен совпадать или быть совместимым с типом возвращаемого значения метода.

Тело метода представляет собой последовательность предложений, заключенных в фигурные скобки. Предложения тела метода объявляют локальные данные, необходимые для работы метода, и кодируют алгоритм, манипулирующий этими данными.

На рис. 1.3 приведена структура кода метода для случая, когда метод не имеет возвращаемого значения.

```
<модификаторы> void <имя метода>(<входные параметры>)
   {
   <предложения тела метода>
   }
```

**Рис. 1.3.** Структура кода метода без возвращаемого значения

Если метод не имеет возвращаемого значения, то в его заголовке вместо типа возвращаемого значения, перед именем метода, записывается служебное слово `void`. Обычно в теле метода, который не формирует возвращаемого значения, не используется оператор `return`. Однако в некоторых случаях для безусловного прерывания выполнения метода, можно в его тело включить предложение, состоящее только из служебного слова `return`. Такое предложение является инструкцией на безусловное завершение работы метода.

Метод завершает свою работу в одном из следующих случаев:

- завершено выполнение всех предложений тела метода;
- достигнуто предложение с оператором return;
- произошла исключительная ситуация, препятствующая дальнейшей работе метода.

Язык программирования Java позволяет кодировать, а при выполнении кода контролировать широкий набор классов потенциальных исключительных ситуаций. Во второй части книги изучаются вопросы, связанные с учетом исключительных ситуаций в программном коде.

Ниже приведены примеры фрагментов заголовков методов, включающие тип возвращаемого значения, имя метода и списки входных параметров. Приведенные ниже фрагменты заголовков методов фигурировали в примере объявления класса на рис. 1.1.

```
void setName(String newName)
String getGender()
```

Первая строка является частью заголовка метода с именем `setName` и одним входным параметром. Этот параметр имеет имя `newName` и тип `String`. Метод не имеет возвращаемого значения. Сигнатурой метода является: `setName(String)`.

Вторая строка является частью заголовка метода с именем `getGender` у которого нет входных параметров. Метод возвращает значение типа `String`. Сигнатура метода совпадает с его именем.

В левой части заголовка метода располагаются служебные слова-модификаторы, специфицирующие общие свойства метода. Изучению общих свойств методов посвящены разделы второй части книги. Поэтому сейчас перечислим несколько модификаторов методов с минимальными комментариями.

- `package`, `protected`, `public`, `private`. Модификаторы доступа, характеризующие доступность метода. Например, если метод помечен модификатором

доступа `private`, то он доступен только в пределах класса, в котором он объявлен, а если метод помечен модификатором `public`, то он не имеет ограничений доступа и доступен из любой точки программы.

- `abstract`. Модификатором `abstract` помечаются абстрактные методы. В объявлении абстрактных методов отсутствует тело. Предполагается, что абстрактные методы кодируются в одном из подклассов класса, в котором объявлен абстрактный метод.
- `static`. Методы, помеченные модификатором `static`, являются статическими методами, или методами, определяющими поведение класса как отдельной сущности. Статические методы предназначены для работы со статическими полями класса и, следовательно, могут обращаться только к полям класса, помеченным модификатором `static`.
- `final`. Методы, помеченные модификатором `final`, не могут быть переопределены в подклассах.

## 1.3. Исходный код, байт-код, интерпретация байт-кода

Люди создают и совершенствуют языки программирования высокого уровня, такие, например, как Java, для облегчения процесса разработки и понимания программ *человеком*. Языки программирования разрабатываются для удобства человека, а не для удобства компьютера. Предложения, записанные, например, на языке программирования Java, слишком сложны для компьютера. Он их не «понимает» и не может выполнить закодированные в них действия. Процессор компьютера может выполнить простые инструкции, которые называются *примитивными командами*. Каждый тип процессора способен выполнять примитивные команды из некоторого, относительно небольшого, списка команд этого процессора. Поэтому, прежде чем поручить выполнение программы компьютеру, она должна быть представлена в виде последовательностей примитивных команд. Преобразование программы, написанной на языке программирования высокого уровня, в цепочки примитивных команд осуществляет не аппаратный блок компьютера, а программная система, называемая *компилятором*.

Программу, записанную на языке программирования высокого уровня, называют *исходным кодом*, а программу, представленную в виде цепочек примитивных команд, — *машинным* или *исполняемым кодом*. Компилятор может быть устроен таким образом, чтобы он осуществлял полное преобразование исходного кода в исполняемый код. В этом случае, поскольку каждый тип процессора обладает своим собственным списком команд, для каждого типа процессора необходим свой компилятор. В результате система программирования становится *платформозависимой*, поскольку программист при написании программы должен учитывать особенности компилятора, при помощи которого он будет получать исполняемый код. Такой подход используется при компиляции исходных кодов для большинства языков программирования высокого уровня, кроме Java.

Язык программирования Java является *платформонезависимым*. Это означает, что один и тот же исходный код, записанный на языке программирования Java, может быть выполнен на компьютере с любым типом процессора. Независимость от платформы компьютера достигается за счет того, что процесс преобразования исходного Java-кода в исполняемый код осуществляется не за один, а за два этапа. На первом этапе компилятор преобразует исходный код в некоторый промежуточный код, называемый *байт-кодом*. Байт-код может рассматриваться как машинный код для некоторого гипотетического процессора. Байт-код стандартизован и состоит из цепочек команд, каждая из которых легко превращается в некоторое количество примитивных команд того или иного процессора. Этот процесс может быть совмещен с процессом выполнения программы и называется интерпретация. *Интерпретатор* последовательно считывает команду байт кода, преобразует ее в одну или несколько примитивных команд и передает процессору на выполнение. Затем перечисленные действия повторяются. Ясно, что интерпретаторов байт-кода нужно столько, сколько существует типов процессоров. Однако интерпретаторы байт-кода для конкретных процессоров значительно проще, чем компиляторы, и их написание относительно несложная задача. Интерпретатор байт-кода для конкретного процессора часто называют *виртуальной машиной Java*.

Рис. 1.4 иллюстрирует двухэтапный процесс преобразования исходного кода на языке программирования Java в исполняемый код конкретного процессора. На первом этапе исходный код компилируется унифицированным Java-компилятором в платформонезависимый байт-код, а на втором — платформозависимая виртуальная машина Java формирует примитивные команды и передает их на выполнение конкретному типу процессора.

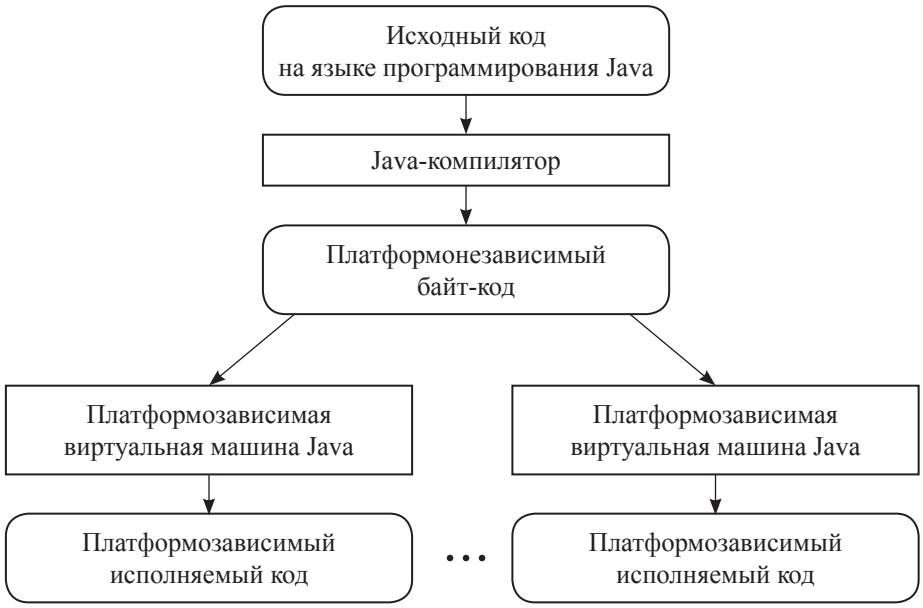



**Рис. 1.4.** Двухэтапный процесс преобразования исходного кода в исполняемый код

Только в простейшем случае исходный код на языке программирования Java имеет вид «цельного куска», хранимого в одном файле. Обычно, программа представляет собой некоторое количество фрагментов, которые разрабатываются и компилируются в байт-код независимо (часто различными программистами). Даже простейшие программы часто используют фрагменты заранее разработанного (предопределенного) байт-кода, входящего в систему программирования Java. Поэтому перед выполнением программы необходим этап *сборки* или *компоновки* отдельных фрагментов байт-кода в единый байт-код. Для системы программирования Java процесс сборки полностью автоматизирован, и программист обычно не принимает в нем участия.

Предопределенные элементы, используемые на этапе сборки, объединены в библиотеку, называемую Java API (Application Programming Interface).

## Упражнения для самостоятельной работы

1.1. Определите понятие «парадигма программирования». Какие парадигмы программирования наиболее популярны в современном программировании?

1.2. Какая из парадигм программирования чаще других используется при разработке коммерческих проектов? Обоснуйте свой ответ.

1.3. Назовите характерные черты императивной парадигмы программирования. Что в терминологии императивного программирования понимают под примитивными данными, выражениями и операторами? Приведите примеры операторов.

1.4. Назовите основное оличие процедурной парадигмы программирования от императивной парадигмы. Дайте определение терминам «программная процедура» и «библиотека процедур».

1.5. На каких утверждениях основана технология структурного программирования? Перечислите основные алгоритмические структуры. В чем состоит преимущество их использования в алгоритмах?

1.6. Опишите специфику организации вычислений в языках декларативного программирования. Приведите примеры декларативных языков программирования.

1.7. Дайте характеристику особенностей логической парадигмы программирования. Каким образом логическое программирование используется в системах искусственного интеллекта?

1.8. Назовите отличия императивной и функциональной модели вычислений. Когда следует использовать функциональную парадигму программирования? Что в функциональных языках реализуется в виде рекурсивной функции?

1.9. Перечислите основные понятия объектно-ориентированной парадигмы. Приведите примеры преимуществ объектно-ориентированной парадигмы в сравнении с другими парадигмами.

1.10. Опишите, как в языке программирования Java объявляются классы. Перечислите структурные элементы класса. Приведите пример объявления класса.

1.11. Дайте определение программного метода. Каковы правила объявления и вызова методов в Java? Что такое сигнатура метода? Чем отличается метод-конструктор от обычного метода?

1.12. Определите понятия «исходный код» и «исполняемый код». Поясните основные отличия процесса компиляция и интерпретации.

1.13. Объясните, за счет чего достигается платформонезависимость языка программирования Java? Дайте определение виртуальной машины Java.

Раздел 2

# ЛЕКСИЧЕСКИЕ ЭЛЕМЕНТЫ

Поскольку одним из главных предметов нашего внимания является исходный код, или текст программы на языке программирования, начнем изучение императивного программирования со знакомства с лексическими основами языка программирования Java. Исходный код представляет собой текст на искусственном языке, который подчиняется строгим лексическим и синтаксическим правилам. Одной из задач компилятора является анализ текста программы и проверка его соответствия лексическим и синтаксическим правилам Java. Компилятор может сформировать корректный байт-код только в том случае, если в тексте программы отсутствуют лексические или синтаксические ошибки.

## 2.1. Лексические элементы программного кода

Исходный текст программы состоит из отдельных структурных элементов, называемых *предложениями*. Каждое предложение, в свою очередь, состоит из некоторого количества *лексических элементов*, являющихся аналогами слов в предложении на естественном языке.

Одним из наиболее важных лексических элементов является *оператор*. Операторы имеют наименования. Например, оператор инкремента, оператор присваивания, оператор преобразования типа и другие. В предложении оператор обозначается одним или несколькими символами из тех, которые имеются на клавиатуре компьютера. Например, для обозначения оператора, выполняющего арифметическую операцию сложения, используется символ «+». Оператор — это инструкция или команда, кодирующая действие, которое необходимо выполнить компьютеру. Действия или операции, которые кодируются при помощи операторов, выполняются над *операндами*. Операнды в предложении также представлены лексическими элементами. Лексические элементы, которые используются в качестве операндов, называются *переменными* и *литералами*. Переменная — это данное, значение которого может изменяться при работе программы, а литерал — это данное, значение которого остается неизменным. Переменная всегда имеет имя. В некоторых случаях литерал тоже может иметь имя. После завершения работы оператора формируется результат выполнения операции.

Предложение может включать лексические элементы, называемые *служебными словами*. Служебное слово — это зарезервированное слово языка программирования Java. Служебные слова могут идентифицировать оператор или все предложение. Например, предложение, при помощи которого кодируется создание объекта, включает служебное слово `new`. Служебные слова нельзя использовать для наименования переменных методов, а также полей и методов классов.

Предложение включает также лексические элементы, являющиеся знаками пунктуации. Эти лексические элементы часто называют разделителями. *Разделители* — это символы, которые используются для разделения лексических элементов, предложений и блоков предложений. Например, для разделения лексических элементов можно использовать символ «пробел» и другие разделители, а для разделения предложений всегда используется символ «точка с запятой».

Текст программы может включать лексический элемент, называемый *комментарием*. Комментарии бывают различных видов и обычно используются для пояснения текста программы. Поэтому комментарии, как правило, представляют собой естественно-языковые вставки в программный код.

Операторы, переменные, литералы, служебные слова и разделители называются лексемами. *Лексемы* являются теми лексическими элементами, которые различаются компилятором и используются при создании байт-кода. Комментарии и некоторые разделители не относятся к лексемам, не нужны для создания байт-кода и поэтому исключаются компилятором из процесса компиляции. На одном из начальных этапов своей работы компилятор выполняет сканирование текста программы с целью выявления *лексем*.

### 2.1.1. Набор символов

Лексические элементы предложения представляют собой последовательности символов и знаков. В большинстве случаев для записи предложений текста программы программисту достаточно тех символов, которые нанесены на клавиатуру компьютера, однако язык программирования Java позволяет использовать значительно более широкий набор символов. Отличительной особенностью языка программирования Java является использование набора символов и знаков, которые представимы в системе кодирования знаков, получившей наименование Unicode.

Система Unicode позволяет кодировать *более миллиона различных знаков* и включает все символы, которые люди использовали для записи и сохранения информации за всю свою историю, начиная от древних систем письменности и по сегодняшний день. Система Unicode включает также большое количество специальных знаков, не относящихся к письму. Кроме того, пространство знаков в Unicode организовано таким образом, что в нем зарезервированы кодовые комбинации, которые сегодня являются «пустыми», но могут понадобиться в будущем. Использование пространства знаков системы Unicode позволяет при

кодировании программ наряду с английским языком использовать любые национальные языки не только для записи строковых литералов и комментариев, но и для записи лексических элементов. Таким образом, например, в тексте программы на языке Java могут использовать имена переменных на любом национальном языке. В Интернете можно найти примеры кодов с использованием, например, китайских иероглифов. Однако, общепринятый стиль программирования предполагает использование исключительно английского языка для записи текста программы. Национальные языки используются, как правило, для записи комментариев и строковых литералов, которые, например, выводятся на экран монитора.

Система Unicode предполагает, что любой символ или знак кодируется при помощи четырехразрядного шестнадцатеричного числа, называемого Unicode-номером. Unicode-номер хранится в памяти компьютера в виде 16-битного двоичного числа. В тексте программы код знака записывается в виде *escape-последовательности*, которая представляет собой Unicode-номер, предваряемый префиксом «`\u`». Например, escape-последовательность строчной буквы «а» латинского алфавита равна «`\u0061`». Ясно, что запись символов в виде escape-последовательности является громоздким и малоудобным способом представления символов, однако этот способ является единственным способом включения в текст программы символов, которые отсутствуют и на клавиатуре компьютера, и в таблице символов текстового редактора. Например, только таким способом можно ввести в текст программы символы старославянского языка. Более подробно система Unicode рассматривается в подразделе 3.3 первой части книги.

### 2.1.2. Комментарии

Текст программы можно комментировать. Обычно, комментарий представляет собой краткий текст на естественном языке, который поясняет те элементы кода, понимание которых непосредственно не следует из анализа предложений программы и затруднено. Грамотные комментарии являются признаком хорошего стиля программирования. Комментарии нужны человеку, но не нужны компьютеру и игнорируются компилятором. Поэтому количество комментариев, которыми снабжен текст программы, никак не влияет на качество байт-кода программы.

В комментарии могут быть включены не только символы, имеющиеся на клавиатуре, но и любые знаки из пространства знаков Unicode. Для этого в том месте комментария, куда должен быть включен специальный знак, записывается его escape-последовательность. Например, в комментарий может быть включен китайский символ, называемый «инь и янь» (символ женского и мужского начала), или символ денежной единицы Северной и Южной Кореи, называемый «вонг». Escape-последовательность символа «инь и янь» записывается в виде «`\u262f`», а escape-последовательность символа «вонг» — «`\u20a9`».

В языке программирования Java предусмотрены три вида комментариев, приведенные в таблице на рис. 2.1.

| Наименование комментария | Способ включения комментария в текст | Описание комментария |
|---|---|---|
| Строчный комментарий | `// комментарий` | Начинается с символов `//` и длится до конца текущей строки. Этот вид комментария обычно используется для комментирования строки. |
| Блочный комментарий | `/* комментарий */` | Располагается между символами `/*` и `*/`. Этот вид комментария используется в тех случаях, когда текст комментария занимает несколько строк. |
| Комментарий документирования | `/** комментарий */` | Располагается между символами `/**` и `*/`. Этот вид комментария размещается перед объявлением класса и/или его членов. Он предназначен для пользователей класса и его можно использовать для автоматического создания справочной документации. |

**Рис. 2.1.** Виды комментариев в языке программирования Java

*Строчный комментарий* является одним из наиболее часто используемых видов комментария. В примерах фрагментов кода, приведенных как в первой, так и во второй частях учебного пособия, использовано большое количество строчных комментариев, которые поясняют код и помогают его описывать в тексте. Например, в четвертом разделе книги, посвященном изучению строковых данных, приведен фрагмент кода

```
String bestCity = "Одесса";                                    // предложение 1
String bestCity = "\u041E\u0434\u0435\u0441\u0441\u0430"; // предложение 2
String bestCity = "Одесс\u0430";                               // предложение 3
```

Фрагмент иллюстрирует различные способы записи одного и того же строчного литерала, который присваивается переменной с именем `bestCity`. Строчные комментарии, введенные в этот фрагмент, позволяют при описании фрагмента кода ссылаться на каждое из предложений в отдельности.

Строчный комментарий можно использовать для временного «отключения» предложений кода при отладке программы. Для этого достаточно в начале каждой строки того фрагмента кода, который необходимо временно «отключить», записать пару символов «//». При компиляции исходного текста в байт-код эти строки будут восприняты как комментарии и пропущены компилятором.

*Блочный комментарий* обычно размещается перед участком кода, который необходимо прокомментировать. Например, в тексте программы перед объявлением

класса можно разместить общие сведения о классе, адресованные пользователю класса и называемые *контрактом класса*. В контракт класса может быть включен перечень методов класса, сведения, достаточные для их использования, и другая информация. Блочный комментарий не обязательно размещать в начальной части программы и использовать для описания контракта. Он может быть размещен в любой части текста программы и иметь любое содержание.

В блочный комментарий нельзя вкладывать другой блочный комментарий. В случае необходимости, нужно просто дополнить текст комментария новыми предложениями. Символы блочного комментария должны располагаться между одной парой открывающих символов «/*» и одной парой закрывающих символов «*/». Например, следующий блочный комментарий записан с ошибкой, которая будет обнаружена компилятором.

```
/* этот метод написать после практики
/* посмотреть алгоритм в 3-ем томе Кнута */
и сдать до конца учебного года */
```

*Комментарий документирования*, так же, как и блочный комментарий, часто используется для описания контракта класса, однако комментарий документирования отличается от блочного комментария тем, что он может быть использован системой программирования Java для автоматического создания справочной документации.

## 2.2. Лексемы

Лексические элементы — это «слова» языка программирования, из которых составляются предложения программного кода. Человек при анализе текста программы воспринимает все лексические элементы. Компилятор — только некоторые. Лексемы — это те «слова», которые «читает» компилятор и использует их для создания байт-кода. Один из первых этапов компиляции называется *лексическим анализом*. Задачей лексического анализа является выделение лексем из исходного текста программы и их распознавание.

Разделителем между лексемами часто является символ пробела, который делает предложение программного кода похожим на предложение естественного языка и облегчает понимание текста программы человеком. Однако некоторые из этих разделителей не нужны компьютеру. Поэтому в процессе выявления лексем компилятор исключает из текста лишние символы пробела, а также все виды комментариев. Удаление пробелов из исходного текста программы на этапе лексического анализа выполняется не механически. Для компилятора «лишними» являются только пробелы между лексемами, но не пробелы внутри лексем. Поэтому компилятор должен уметь отличать «лишние» пробелы от нужных. К лексемам относятся следующие лексические элементы: переменные, литералы, операторы, служебные слова и разделители.

### 2.2.1. Переменные и литералы

Данные в тексте программы представляются двумя видами лексем, называемыми переменными и литералами. Прежде чем рассматривать лексические правила записи переменных и литералов, рассмотрим, какой смысл следует вкладывать в понятия «переменная» и «литерал».

По отношению к алгоритму понятие переменной в программировании аналогично понятию алгебраической переменной. В этом смысле переменная — это данное, которое в процессе работы программы изменяет свое значение. Литерал — это данное, которое в процессе работы программы остается неизменным.

По отношению к архитектуре компьютера переменная может ассоциироваться с ячейкой основной памяти компьютера, в которой хранится данное и содержимое которой меняется в процессе работы программы. Литерал может ассоциироваться с ячейкой памяти, содержимое которой не меняется в процессе работы программы.

Алгебраическая переменная обладает только именем. Компьютерная переменная, в отличие от алгебраической переменной, обладает именем и типом. Поэтому предложение, при помощи которого объявляется переменная, должно включать *имя переменной* и *имя типа переменной*. По отношению к архитектуре компьютера имя переменной можно рассматривать как *символический адрес* ячейки памяти, в которой хранится значение переменной, а *тип переменной определяет размер* этой ячейки. Литерал всегда обладает типом и, в некоторых случаях, может иметь имя. Если литерал обладает именем, то он называется *поименованным литералом*. По отношению к архитектуре компьютера имя и тип литерала интерпретируются точно так же, как имя и тип переменной.

По отношению к типу переменных все переменные принято делить на две группы: *простые переменные* и *переменные ссылочного типа*. Характерной особенностью простых переменных является фиксированный размер ячейки памяти, которая предназначена для хранения значения переменной. Поэтому с простыми переменными связывают такие данные, которые можно хранить в ячейках фиксированного размера. Типы этих данных называют *примитивными типами*. В языке программирования Java имеется 8 типов примитивных данных. Например, переменная примитивного типа `int` хранится в ячейке размером в 4 байта или 32 бита. Тип `int` используется для представления целочисленных переменных.

Переменные ссылочного типа предназначены для работы с программными объектами. Размеры объектов различных классов не могут быть фиксированы. Например, размер объекта класса `String`, который используется для представления строки символов, зависит от количества символов в строке. Поэтому для хранения объектов в памяти компьютера выделяется участок памяти «по потребности», а переменная ссылочного типа хранит не сам объект, а ссылку на него или адрес участка памяти, где находится сам объект.

Таким образом, в ячейке памяти, отведенной для хранения простой переменной, хранится само данное, а в ячейке памяти, отведенной для хранения переменных ссылочного типа — ссылка на программный объект. Ссылочные переменные

имеют тип. Имя типа ссылочной переменной, в большинстве случаев, является именем класса, на объект которого эта ссылочная переменная хранит ссылку.

Будем различать глобальные и локальные переменные. *Глобальные переменные* — это, по сути, поля класса. Глобальные переменные объявляются в блоке класса и доступны всем методам этого класса. Это означает, что в предложениях любого метода данного класса можно использовать имя поля класса как для чтения значения поля, так и для записи в поле нового значения. Если в программе не предусмотрены специальные меры по инициализации глобальных полей (например, при помощи конструктора), то это осуществляется автоматически, «по умолчанию».

*Локальные переменные* — это переменные, которые объявленны в блоке метода и доступны только из предложений данного метода. Локальные переменные не инициализируются «по умолчанию», и программист должен предусмотреть их инициализацию. Если в теле метода отсутствует инициализация локальных переменных, то это воспринимается компилятором как синтаксическая ошибка.

В предложениях программного кода имена переменных или поименованных литералов представлены строками символов. При формировании имен переменных и литералов необходимо придерживаться следующих правил: (1) имя всегда начинается с буквы; (2) в имени могут использоваться только буквы, цифры, символ подчеркивания и символ доллара; (3) в имени нельзя использовать символ пробела; (4) строчные и прописные буквы различаются компилятором и рассматриваются им как различные символы. В языке программирования Java для записи имен переменных и литералов, а, вообще говоря, имен всех программных сущностей (например, имен полей или методов классов) можно использовать любые национальные языки. Однако, как уже было отмечено, общепринятый стиль программирования предполагает использование для этой цели английского языка. От выбора имен переменных зависит восприятие и понимание текста программы. Имя переменной должно отражать ее сущность. Например, переменной, предназначенной для хранения значение радиуса окружности лучше дать имя `radius`, а не `x`. Несмотря на то, что в алфавите английского и русского языков имеются совпадающие буквы (например, `E`, `C` и другие), при записи имени переменной необходимо использовать только буквы английского алфавита. Длина имени формально не ограничена. Тем не менее, не следует использовать имена, состоящие из большого количества символов. Слишком пространные имена затрудняют восприятие текста и являются источником синтаксических ошибок.

Имена типов простых переменных зарезервированы и являются служебными словами.

### 2.2.2. Операторы

Изменение значений локальных переменных осуществляется путем выполнения операций. Для кодирования операций в тексте программы используются лексические элементы, называемые операторами. Входные данные оператора называются

операндами. Один оператор может выполнять операцию, используя один или несколько операндов.

По отношению к количеству используемых операндов операторы делятся на: *унарные*, *бинарные* или *тернарные*. Унарные операторы выполняют действие над одним операндом, бинарные — над двумя, а тернарные — над тремя. После завершения операции оператор возвращает результат операции. Унарными операторами являются, например, операторы инкремента и оператор логического отрицания. Оператор инкремента увеличивает целочисленное значение своего единственного операнда на единицу, а оператор логического отрицания изменяет значение своего единственного операнда, булевского типа, на противоположное. Примерами бинарных операндов являются операторы, предназначенные для выполнения арифметических операций сложения, вычитания, умножения и деления. В языке программирования Java имеется только один тернарный оператор, называемый «условный оператор ?».

Оператор часто обозначается одним или несколькими символами. Для обозначения операторов, предназначенных для выполнения арифметических и логических операций, а также операций сравнения используются символы, схожие с соответствующими математическими символами.

В языке программирования Java имеется большое количество операторов, многие из которых будут изучаться в соответствующих разделах книги. Трудно и, возможно, нецелесообразно вводить всеобъемлющую классификацию операторов языка программирования Java, однако в рамках императивного программирования наиболее значимыми являются следующие пять групп операторов.

- Операторы арифметических операций.
- Операторы поразрядных операций.
- Операторы сравнения.
- Операторы логических операций.
- Оператор присваивания.

Дадим краткую характеристику операторов арифметических, логических и поразрядных операций, а также операторов сравнения. Подробному изучению этих операторов посвящены последующие разделы книги.

Большинство *операторов арифметических операций* являются бинарными и предназначены для кодирования операций, выполняемых над двумя операндами. Операндами арифметических операций являются переменные, которые объявлены с типом, соответствующим числовым данным, либо числовые литералы. В некоторых случаях арифметические операции можно выполнять над данными, которые можно рассматривать как целые числа.

*Операторы поразрядных операций* предназначены для выполнения операций с битовыми последовательностями, которые интерпретируются как целые числа. Среди операторов этой группы есть унарные и бинарные операторы. Например, при помощи унарных операторов поразрядных операций можно выполнить поразрядный сдвиг битовой последовательности вправо или влево. Использование этих операторов ограничено рамками специфических методов, и они редко используются при изучении императивного программирования.

*Операторы сравнения* предназначены для сравнения величин двух операндов и поэтому являются бинарными. Оператор сравнения можно ассоциировать с «да/нет» вопросом естественного языка. При помощи оператора сравнения можно выяснить, например, является ли величина первого операнда больше, чем величина второго. Возвращаемым значение оператора сравнения является один из литералов булевского типа `true` или `false`. Если оператор сравнения возвращает значение `true`, то это означает, что ответ положительный, а если — `false` –то ответ отрицательный.

*Операторы логических операций* предназначены для кодирования основных логических операций над переменными булевского типа: «операция отрицания», «операция И» и «операция ИЛИ». Эти операции выполняются по тем же правилам, что и операции «отрицание», «конъюнкция» и «дизъюнкция» в пропозициональной логике.

*Оператор присваивания* является одним из наиболее употребительных операторов императивного программирования. Рассмотрим его подробнее, чем предыдущие операторы, после того, как введем понятие выражения.

*Программное выражение* позволяет объединить несколько операторов и их операндов в одном предложении и может рассматриваться как аналог алгебраического выражения. Таким образом, выражение является закодированной последовательностью операций. Лучшим способом задания последовательности операций в выражении является использование круглых скобок, как это делается в арифметике или алгебре. Однако, если даже в выражении опущены скобки, компьютер сам определит, в какой последовательности нужно выполнять операции в соответствии с их приоритетами. Выражение обладает возвращаемым значением, которое представляет собой данное, образовавшееся в результате выполнения последовательности операций, закодированных в выражении. Возвращаемое значение обладает типом. Тип возвращаемого значения выражения определяется семантикой его операторов и типами операндов. Язык программирования Java позволяет использовать разнотипные данные в одном выражении. В некоторых случаях, когда это не приводит к ошибке, компилятор осуществляет неявное преобразование типов операндов и сам определит тип возвращаемого значения. Однако, программист должен знать правила, которым руководствуется компилятор при определении этого типа.

В наиболее общей форме оператор присваивания имеет структуру, приведенную на рис. 2.2.

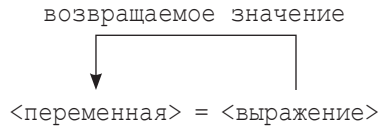



**Рис. 2.2.** Структура оператора присваивания

Символом оператора присваивания является знак равенства. Слева от знака равенства специфицируется переменная. Если переменная уже объявлена в

предыдущих предложениях кода, то спецификацией является имя переменной. Если переменная еще не объявлена, то она специфицируется указанием типа и имени. Справа от знака равенства записывается выражение.

Оператор присваивания кодирует следующие действия. Вначале вычисляется возвращаемое значение выражения, которое затем записывается в переменную, специфицированную в левой части оператора. Тип возвращаемого значения выражения и тип переменной должны быть одинаковыми или совместимыми.

Структура оператора присваивания, приведенная на рис.2.2, является наиболее общей. В частном случае, справа от знака равенства может располагаться имя ранее объявленной и проинициализированной переменной либо литерал. Если в правой части записано имя переменной, то оператор присваивания кодирует действие, заключающееся в записи в переменную левой части значение переменной из правой части. В результате выполнения этого действия, значение переменной в левой части будет обновлено, и обе переменные будут хранить одинаковые значения. Если в правой части записан литерал, то оператор присваивания кодирует действие, заключающееся в записи литерала, в переменную в левой части. На рис. 2.3 приведены примеры предложений с оператором присваивания.

```
circleArea = pi * (radius * radius);        // предложение 1
float circleArea = pi * (radius * radius);  // предложение 2
circleArea1 = circleArea2;                  // предложение 3
float pi = 3.14159f;                        // предложение 4
```

**Рис. 2.3.** Примеры предложений с оператором присваивания

Первое предложение на рис. 2.3 иллюстрирует случай, когда в левой части оператора присвивания записано имя ранее объявленной переменной `circleArea` (площадь круга), а в правой — выражение. Выражение кодирует формулу нахождения площади круга при известном радиусе и составлено из переменных с именами `pi` и `radius` и операторов арифметической операции умножения в виде символа «*». В результате выполнения действий, закодированных в первом предложении, для заданного радиуса (значение переменной `radius`) будет вычислена площадь круга, после чего вычисленное значение площади круга будет записано в переменную `circleArea`.

Второе предложение отличается от первого только тем, что в левой части специфицируется переменная, которая ранее не была объявлена. Поэтому указывается не только имя переменной, а и ее тип `float`. Переменные типа `float` предназначены для хранения вещественных чисел. Второе предложение совмещает объявление переменной и ее инициализацию при помощи оператора присваивания.

Третье предложение иллюстрирует случай, когда и в левой, и в правой частях оператора присваивания записаны имена переменных. Предполагается, что переменная с именем `circleArea1` ранее объявлена, но не обязательно проинициализирована, а переменная с именем `circleArea2` должна быть ранее объявлена и

проинициализирована. В результате выполнения третьего предложения в обеих переменных будет записано значение переменной `circleArea2`.

Четвертое предложение иллюстрирует случай, когда в левой части объявлена переменная с именем `pi`, а в правой части — литерал типа `float`. После выполнения четвертого предложения переменная с именем `pi` будет проинициализирована значением литерала.

### 2.2.3. Служебные слова

Лексемы, называемые служебными словами, не могут использоваться как имена переменных, полей, методов, классов и других программных сущностей, поскольку правила их употребления строго регламентированы в самом языке. На рис. 2.4 приведен перечень служебных слов языка программирования Java.

```
abstract    continue    for         new         switch
assert      default     goTo        package     synchronized
boolean     do          if          private     this
break       double      implements  protected   throw
byte        else        import      public      throws
case        enum        instanceof  return      transient
catch       extends     int         short       try
char        final       interface   static      void
class       finally     long        strictfp    volatile
const       float       native      super       while
```

**Рис. 2.4** Перечень служебных слов языка программирования Java

Значения некоторых литералов внешне выглядят как служебные слова, но к ним не относятся. Например, к служебным словам не относятся литералы `true` и `false` (литералы данных типа `boolean`), а также литерал `null` (литерал данных ссылочного типа). Однако эти слова также нельзя использовать в качестве имен переменных и других программных сущностей.

### 2.2.4. Разделители

Лексемы, называемые разделителями, используются для явного отделения одних лексем от других. Разделители помогают компилятору выделять и идентифицировать лексемы при лексическом анализе. В качестве разделителей в тексте программы используются одиночные символы (например, точка или запятая) или парные символы (например, круглые или квадратные скобки). В таблице на

рис. 2.5 приведены разделители, используемые в языке программирования Java. Не все, что написано во второй колонке таблицы на рис. 2.5, может быть понятно на начальных этапах изучения программирования на Java. В последующих разделах книги назначения разделителей подробно описаны и проиллюстрированы примерами.

| Символ разделителя | Наименование и назначение разделителя |
|---|---|
| `;` | Разделитель «точка с запятой» используется для разделения предложений. Каждое предложение должно завершаться этим разделителем |
| `,` | Разделитель «запятая» используется для: (1) разделения имен однотипных переменных в том случае, когда они объявляются в виде списка в одном предложении; (2) разделения элементов списка формальных и фактических параметров методов и конструкторов при их объявлении и вызове |
| `.` | Разделитель «точка» используется для разделения элементов составного имени во всех случаях использования составных имен. Например, для отделения имени класса от имени пакета, в котором размещен класс, или в сообщении для отделения имени объекта от имени вызываемого метода. |
| `:` | Разделитель «двоеточие» используется для отделения метки от предложения или блока предложений. |
| «`пробел`» | Разделитель «пробел» используется внутри лексем и между лексемами. Разделитель может размещаться между лексемами для удобства восприятия текста программы человеком. Этот разделитель не учитывается компилятором. Разделитель внутри лексемы используется компилятором. Например, в лексеме «`return 0`» пробел обязателен. |
| `( )` | Разделитель «круглые скобки» используется: (1) для указания последовательности выполнения операций в выражениях; (2) для отделения списка формальных или фактических параметров метода от имени метода; (3) в операторе явного преобразования типа; (4) в операторах, изменяющих естественный порядок выполнения предложений, например, в операторе `if`. |
| `{ }` | Разделитель «фигурные скобки» используется для отделения блока предложений от остальных предложений кода |
| `[ ]` | Разделитель «квадратные скобки» используется при объявлении и работе с массивами данных. |

**Рис. 2.5.** Перечень разделителей языка программирования Java

При использовании разделителей-скобок необходимо записывать и левую, и правую скобки. Это парные символы, которые предназначены для указания начала и конца выделенной группы лексем. Если, например, пропущена правая скобка, то компилятор расценит это как синтаксическую ошибку на этапе лексического анализа.

## Упражнения для самостоятельной работы

2.1. Определите понятие «лексема». Перечислите все виды лексем в языке Java. Объясните, почему комментарии не относятся к лексемам.

2.2. Назовите систему кодирования символов и знаков, применяемую в языке программирования Java. В каком виде хранится в памяти компьютера код символа или знака? Что представляет собой `escape`-последовательность? Запишите `escape`-последовательность для строчной буквы «`a`».

2.3. Объясните, зачем в тексте программы применяются комментарии. Перечислите все виды комментариев. Сколько комментариев в следующем примере программного кода?

```
int x = 0; /* text // text /*
int y=1; // text */ // text */
```

2.4. Дайте определение переменной. Назовите сходства и отличия понятий переменной в математике и программировании.

2.5. Объясните, в чем состоит отличие между примитивными и ссылочными переменными. Как хранятся эти переменные в памяти компьютера?

2.6. Дайте определение локальным и глобальным переменным в Java. Поясните, в чем состоит отличие в их инициализации?

2.7. Перечислите правила записи имен переменных, установленные в Java. Какие из указанных ниже последовательностей символов могут быть использованы в качестве имен переменных?

```
rate1, 1stPlayer, myprogram.java,
this, TimeLimit, numberOfWindows,
miles, Test, a++, --a, 4#R, $4,
#44, apps, class, public, int, radius
```

2.8. Может ли программа, написанная на языке программирования Java, содержать две различные переменные с именами `number` и `Number`?

2.9. Текущее значение целочисленной переменной с именем `weight` равно 100. Чему будет равно значение этой переменной после выполнения следующего оператора присваивания: `weight = -25`? Объясните, почему?

2.10. Текущее значение целочисленной переменной с именем `diameter` равно 5. Как изменится значение этой переменной после выполнения следующего оператора присваивания: `diameter = diameter*4`? Объясните, почему?

2.11. Запишите оператор присваивания, который установит значение переменной с именем `distance`, равным значению переменной с именем `time`, умноженному на 80. Все переменные имеют тип `int`.

2.12. Запишите оператор присваивания, который установит значение переменной с именем `interest`, равным значению переменной с именем `balance`, умноженному на значение переменной с именем `rate`. Примем, что все переменные имеют тип `int`.

2.13. Пусть даны две целочисленные переменные с именами `a`, `b`. Составьте фрагмент программы, после выполнения которого значения этих переменных поменяются местами (новое значение переменной с именем `a` будет равно старому значению переменной с именем `b` и наоборот).

2.14. Решите упражнение 2.13, не используя дополнительных переменных и предполагая, что значениями переменных могут быть произвольные целые числа.

Раздел 3

# ПРИМИТИВНЫЕ ТИПЫ ДАННЫХ

Способность выполнять арифметические и логические операции над данными является фундаментальной функцией современных компьютеров, унаследованной ими еще от вычислительных машин первого поколения.

В некоторых объектных языках программирования, например в Smalltalk, отсутствует понятие данное, а числа рассматриваются как объекты соответствующих классов. Ничего не мешает ввести, например, понятие класса целых чисел и определить общие свойства и методы этого класса. Тогда любое целое число может рассматриваться как объект класса целых чисел. Язык программирования Java является гибридным в том смысле, что в нём различаются понятия примитивных данных и объектов. Объекты могут обладать поведением и выполнять некоторую работу при помощи своих методов, но над объектами нельзя выполнять арифметические или логические операции.

Примитивные данные всегда пассивны и не обладают собственным поведением. Над примитивными данными выполняются арифметические и логические операции. К примитивным данным языка программирования Java относятся: *целочисленные данные*, *данные для представления вещественных чисел*, *логические данные и данные для представления символов*.

Размер участка памяти, который выделяется для хранения объекта, не может быть фиксированным и определяется количеством и типами полей, количеством и размерами методов, конструкторов и вложенных объектов. Отличительной особенностью примитивных данных является фиксированный объём основной памяти, который выделяется для их хранения. Этот объём соответствует типу данных. Поэтому имя типа примитивных данных может использоваться для определения объёма основной памяти, выделяемого для его хранения. Например, для хранения любого целочисленного данного, имеющего тип `int`, всегда выделяется 4 байта или 32 бита памяти.

## 3.1. Типы данных для представления чисел

В языке программирования Java имеется четыре типа данных для представления целых чисел и два типа данных для представления вещественных чисел.

Для обозначения того факта, что некоторое поле класса или локальная переменная метода предназначены для хранения целого числа (целочисленного данного), используется один из типов, обозначаемых служебными словами

```
byte, short, int, long
```

При движении слева направо, по приведенному списку, размер памяти, выделяемый для хранения целого числа, увеличивается ровно вдвое. Ясно, что чем больше участок памяти, который выделяется для хранения целого числа, тем больше диапазон чисел, которые могут храниться в этом участке памяти. В таблице на рис. 3.1 показано соответствие между типом целочисленного данного, диапазоном хранимого числа и размером памяти.

Целочисленные данные одного из типов `byte`, `short`, `int`, `long` нужны, главным образом, для счета и манипулирования числами, полученными *в результате счета*. Если, например, мы хотим подсчитать общее количество студентов, слушающих лекцию на потоке, то мы должны просуммировать количество студентов каждой из групп, включенных в поток. Ясно, что целые числа практически неприменимы в тех случаях, когда необходимо хранить *результаты измерений* и манипулировать ими. Крайне редко результатом измерений является целое число. Как правило — это вещественное число, имеющее целую и дробную части. Чем точнее выполняется измерение, тем больше значащих цифр содержит вещественное число. С увеличением точности представления результата измерения, как правило, увеличивается количество значащих цифр после запятой.

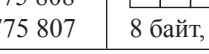
| Тип | Диапазон чисел | Размер памяти |
|---|---|---|
| `byte` | – 128<br>+ 127 | <br>1 байт, 8 бит |
| `short` | – 32 768<br>+ 32 767 | <br>2 байта, 16 бит |
| `int` | – 2 147 483 648<br>+ 2 147 483 647 |  <br>4 байта, 32 бита |
| `long` | – 9 223 372 036 854 775 808<br>+ 9 223 372 036 854 775 807 |  <br>8 байт, 64 бита |

**Рис. 3.1.** Типы целочисленных данных и диапазоны их значений

В компьютерных науках вещественные числа часто называются *числами с плавающей запятой*. Наименование отражает общепринятый способ хранения вещественных чисел в основной памяти компьютера. Этот способ заключается в том, что информация о числе хранится в двух частях: (1) информация о значащих цифрах числа (эта часть называется мантиссой) и (2) информация о расположении запятой,

отделяющей целую часть числа от дробной части числа (эта часть называется порядком). Приведем пример, иллюстрирующий понятия мантисса и порядок. Пусть нам необходимо хранить значение числа π с точностью до пятого знака после запятой. Иными словами, нам необходимо хранить число 3,14159. Тогда информация об этом числе представима следующими частями:

(1) 314159 — значащие цифры, или мантисса
(2) –5 — информация о расположении запятой, или порядок (запятая отделяет пять цифр мантиссы, считая справа налево).

Смысл порядка легко понять, если записать исходное число в виде произведения мантиссы на степень числа 10.

$$3{,}14159 = 314159 \times 10^{-5}$$

Для обозначения того факта, что некоторое поле класса или локальная переменная метода предназначены для хранения числа с плавающей запятой, используется один из двух типов, обозначаемых служебными словами

`float`, или `double`

Данные типа `float` используются для хранения и манипулирования числами, мантисса которых содержит, примерно, *восемь* десятичных цифр и, следовательно, данные типа `float` целесообразно использовать в расчетах, когда восемь значащих цифр достаточны для обеспечения необходимой точности. Данные типа `double` обеспечивают в два раза большую точность, чем данные типа `float`, и позволяют хранить числа, мантисса которых содержит, примерно, *шестнадцать* десятичных цифр. В таблице на рис. 3.2 показано соответствие между типом данного с плавающей запятой, диапазоном хранимого числа и размером памяти.

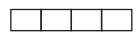
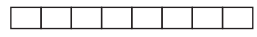
| Тип | Диапазон чисел | Размер памяти |
|---|---|---|
| `float` | Абсолютная величина наибольшего числа равна примерно $3{,}4 \times 10^{38}$<br>Абсолютная величина наименьшего числа равна примерно $1{,}4 \times 10^{-45}$ |  <br>4 байта, 32 бита |
| `double` | Абсолютная величина наибольшего числа равна примерно $1{,}8 \times 10^{308}$<br>Абсолютная величина наименьшего числа равна примерно $2{,}2 \times 10^{-308}$ |  <br>8 байт, 64 бита |

**Рис. 3.2.** Типы данных с плавающей запятой и диапазоны их значений

Для представления чисел типа `float` необходимо четыре байта, ровно столько, сколько используется для представления чисел типа `int`. Однако, принятый способ хранения чисел в виде мантиссы и порядка позволяет существенно расширить диапазон представления чисел типа `float`.

Для представления чисел типа `double` необходимо восемь байт, что значительно увеличивает не только точность, но и диапазон хранимых чисел типа `double`.

### 3.1.1. Литералы числовых данных. Объявление числовых переменных

Литералы типов `byte`, `short` и `int` записываются в предложениях как обычные целые числа. В конце литерала типа `long` записывается символ «`L`». Допустимо также использование не прописной (большой), а строчной (малой) буквы «`l`». Однако, лучше использовать прописную букву, поскольку строчную легко спутать с единицей. На рис. 3.3 приведены примеры записи целочисленных литералов.

```
byte  a = 1;
short b = 2;
int   c = 3;
long  d = 1000L;
```

**Рис. 3.3.** Примеры записи целочисленных литералов

При записи литералов типов `float` и `double` используется несколько эквивалентных нотаций. На рис. 3.4 приведены примеры предложений, в которых все переменные имеют одно и то же значение.

```
float a = 12.3f;    //традиционная запись вещественного числа
float b = 123e-1f;  //мантисса 123, порядок -1
float c = 0.123e2f; //мантисса 0.123, порядок 2
float d = 1.23e1f;  //мантисса 1.23, порядок 1
```

**Рис. 3.4.** Примеры записи литералов с плавающей запятой

Литерал может быть записан способом, принятым в математике для записи вещественных чисел с заменой символа «запятая» на символ «точка» (переменная `a`), либо в форме с плавающей запятой (переменные `b`, `c`, и `d`). Если литерал записан в форме с плавающей запятой, то разделителем между мантиссой и порядком является символ «`e`». Форму записи с символом «`e`» удобно использовать в тех случаях, когда литерал содержит большое количество нулей. Например,

вместо `0.00000000000123` удобнее записать `123e-14`

Если необходимо явно указать тип литерала, то в конце литерала типа `float` записывается символ «`f`», а в конце литерала типа `double` — символ «`d`» (допустимо использование прописных букв):

```
3.14159f   3.14159d
```

Если тип литерала не указан явно, то по умолчанию ему приписывается тип `double`.

Прежде чем использовать переменную в коде, *она должна быть объявлена*. На рис. 3.5 приведены несколько вариантов объявления целочисленных переменных.

```
byte i;              // предложение 1

short i = 8;         // предложение 2

int numb1,           // предложение 3
    numb2,
    numb3;

long numb1 = 100L,   // предложение 4
     numb2 = 150L,
     numb3 = 200L;
```

**Рис. 3.5.** Объявление целочисленных переменных

Предложение 1 иллюстрирует простейший способ объявления. Вначале записывается имя типа (`byte`), а затем имя переменной (`i`). Предложение 2 иллюстрирует совмещение объявления переменной с присваиванием ей начального значения. Справа от символа оператора присваивания может быть записан не только целочисленный литерал, но и выражение, включающее арифметические операции и/или вызов метода, имеющего возвращаемое значение. Важно, чтобы все члены арифметического выражения и/или возвращаемое значение метода были целыми числами соответствующего типа или могли автоматически преобразовываться в тип объявленной переменной без потери значащих цифр. В предложении 2 это тип `short`. Вопросы корректного преобразования типов будут рассмотрены ниже. Предложение 3 иллюстрирует объявление списка переменных (`numb1`, `numb2`, `numb3`) в одном предложении. Предложение 4 иллюстрирует совмещение объявления списка переменных с заданием им начальных значений.

Варианты объявления переменных для представления чисел с плавающей запятой такие же, как и варианты, приведенные, на рис. 3.5. Отличие заключается в имени типа и способе записи литералов. На рис. 3.6 приведены примеры объявления переменных для чисел с плавающей запятой.

```
float radius;

float radius = 28.76f;

double measurement1,
       measurement2,
       measurement3;

double measurement1 = 19445e9,
       measurement2 = 19456e9,
       measurement3 = 19448e9;
```

**Рис. 3.6.** Объявление переменных для представления чисел с плавающей запятой

Локальные переменные не инициализируются автоматически, и, если программист не позаботился о задании начальных значений объявленным локальным переменным, то компилятор выведет сообщение о ошибке.

### 3.1.2. Арифметические операции

Над значениями переменных и/или литералами числовых типов можно выполнять арифметические операции, представляя их в виде выражений примерно так, как это делается в арифметике и алгебре. В таблице на рис. 3.7 приведены символы операторов, используемых для выполнения арифметических операций над числовыми данными.

| Символ оператора | Наименование операции |
|---|---|
| + | сложение |
| - | вычитание |
| * | умножение |
| / | деление |
| % | нахождение остатка от деления |

**Рис. 3.7.** Символы операторов для выполнения арифметических операций над числовыми данными

Хотя символы арифметических операций одинаковы для всех числовых типов данных, необходимо знать специфику их «работы» с операндами, представленными целочисленными типами данных, типами данных с плавающей запятой и смешанными типами данных.

На рис. 3.8 приведен пример фрагмента кода, иллюстрирующий применение операторов умножения и деления к целочисленным данным.

```
short height = 5,
      base = 13;
short triangleArea = (base * height)/2;
```

**Рис. 3.8.** Умножение и деления целочисленных данных

Код, приведенный на 3.8, объявляет переменные для представления основания и высоты треугольника и находит его площадь. Площадь треугольника `triangleArea` равна половине произведения основания `base` на высоту `height`, опущенную на основание. Для приведенных на рис. 3.6 значений, произведение основания на высоту — это целое число, равное 65, а точное значение площади треугольника равно 32,5. Однако результат выполнения арифметических операций, заданных выражением

```
(base * height)/2
```

равен 32. Остаток от деления числа 65 на 2 проигнорирован. Отбрасывание остатка от деления — это специфика выполнения операции деления для целочисленных данных. Этот остаток можно получить при помощи предложения

```
short remainder =(base * height)%2;
```

Операцию нахождения остатка от деления удобно использовать для определения того, является ли значение целочисленной переменной четным или нечетным числом. Для этого необходимо проверить, чему равен остаток от деления значения переменной на два. Остаток от деления четного числа на два равен нулю, а остаток от деления нечетного числа на два больше нуля.

Для того, чтобы получить значение площади треугольника, выражаемое числом 32,5, необходимо все переменные и литералы представить в форме с плавающей запятой. Операция деления в коде, приведенном на рис. 3.9, возвратит значение 32,5.

```
float height = 5.0f,
     base = 13.0f;
float triangleArea = (base * height)/2.0f;
```

**Рис. 3.9.** Умножение и деление данных с плавающей запятой

В предложениях с оператором присваивания и в арифметических выражениях можно использовать переменные и литералы различных типов. Несмотря на то, что во многих случаях компилятор не выведет сообщение об ошибке в случае использования разнотипных данных, необходимо осознанно использовать

возможность «смешения» типов. Будем различать два случая использования разнотипных данных в предложениях: (1) присваивание разнотипных данных в переменную заданного типа; (2) использование разнотипных данных в арифметических выражениях.

Рассмотрим вначале первый случай, когда в левой и правой частях предложения с оператором присваивания находятся данные различных типов. Естественно допустить, что значение переменной или литерала более «короткого» типа может быть присвоено в переменную более «длинного» типа, поскольку в этом случае отсутствует опасность потери информации. Например, значение переменной типа `short` может быть записано в переменную типа `int`, а значение переменной типа `float` может быть записано в переменную типа `double`. Назовем такие присваивания безопасными. *Безопасные присваивания* являются допустимыми в том смысле, что не порождают сообщения об ошибке на этапе компиляции кода. Общий случай допустимых присваиваний иллюстрируется последовательностью на рис. 3.10. Допустимым является присваивание данного любого из типов, расположенных слева, в переменную, тип которой расположен справа.

```
byte → short → int → long → float → double
```

**Рис. 3.10.** Допустимые присваивания числовых данных различных типов

Как следует из последовательности, приведенной на рис. 3.10, допустимыми являются не только присваивания более «коротких» целочисленных типов в более «длинные» целочисленные типы, но и присваивание целочисленных типов в типы с плавающей запятой. В этом общем правиле есть одна особенность, которая относится к присваиванию данного типа `long` в переменную типа `float` или `double`. Целочисленное данное типа `long` может быть представлено с 19 верными значащими цифрами (см. таблицу на рис. 3.1), а количество верных значащих цифр для чисел типа `float` и `double` не превышает 8 и 16, соответственно. Следовательно, хоть присваивание данного типа `long` в переменную типа `float` или `double` является допустимым, оно может привести к потере значащих цифр.

На рис. 3.11 приведен фрагмент кода, иллюстрирующий предложения с допустимыми присваиваниями.

```
long a, b, c;
int i = 10, j = 50;
float x;
   a = 100;  // предложение 1
   b = i;    // предложение 2
   c = 2*j;  // предложение 3
   x = j;    // предложение 4
```

**Рис. 3.11.** Примеры предложений с допустимыми присваиваниями числовых данных

Первое предложение записывает в переменную a типа long литерал типа int, второе — в переменную b типа long значение переменной i типа int, третье — в переменную c типа long результат вычисления выражения типа int, а четвертое в переменную типа float значение переменной типа int.

Ранее мы предположили, что допустимыми являются присваивания «коротких» типов в «длинные» типы. Естественно также предположить, что значение данного более «длинного» типа нельзя присвоить переменной более «короткого» типа, поскольку в этом случае возможна потеря информации. Например, значение переменной типа long нельзя непосредственно записать в переменную типа int, даже если фактическое значение переменной типа long таково, что это не приводит к потере информации. Назовем такие присваивания опасными. *Опасные присваивания* являются недопустимыми, поскольку порождают сообщения об ошибке на этапе компиляции кода. Операции присваивания, приведенные в предложениях на рис. 3.12 являются недопустимыми.

```
long b;
int i = 10, j;
double y;
   i = b;
   j = 100L;
   b = y;
```

**Рис. 3.12.** Примеры предложений с недопустимыми присваиваниями числовых данных

Однако, если нам необходимо, например, присвоить значение данного типа long в переменную типа int и мы уверены, что это не приведёт к потере информации, то такое присваивание можно сделать допустимым в случае использования *оператора явного преобразования типа*. Оператор явного преобразования типа сообщает компилятору, что некоторое данное «длинного» типа может быть записано в переменную «короткого» типа и представляет собой наименование требуемого типа, заключённое в круглые скобки и размещённое непосредственно перед данным, подлежащим преобразованию.

В предложениях, приведенных на рис. 3.13, использован оператор явного преобразования типа, и теперь они являются допустимыми.

```
long b;
int i = 10, j;
double y;
   i = (int)b;    // предложение 1
   j = (int)100L; // предложение 2
   b = (long)y;   // предложение 3
```

**Рис. 3.13.** Примеры использования оператора явного преобразования типа для допустимого присваивания числовых данных

Оператор явного преобразования типа в первом предложении требует преобразовать тип `long` переменной `b` в тип `int`, а затем результат преобразования присваивается в переменную `i`; во втором предложении — преобразовать тип `long` литерала `100L` в тип `int`; в третьем предложении — преобразовать тип `double` переменной `y` в тип `long`.

Особенность «работы» оператора явного преобразования типа заключается в том, что при преобразовании данного с плавающей запятой в данное целочисленного типа она не округляет вещественное число, а просто отбрасывает дробную часть.

При использовании разнотипных данных в арифметических выражениях важно знать ответ на вопрос: «Какой тип получится в итоге и будет возвращен выражением после выполнения всех операций, предусмотренных выражением?». Ответ на этот вопрос определяется правилом выполнения арифметических операций над разнотипными данными, которое гласит, что в результате выполнения операций всегда возвращается значение данного самого «длинного» типа.

На рис. 3.14 приведен пример кода, вычисляющего объем прямоугольного параллелепипеда в виде произведения площади основания на высоту в том случае, когда ширина и длина основания параллелепипеда, а также его высота заданы разнотипными данными.

```
short width = 35;                    // предложение 1
int length = 10;                     // предложение 2
float heigh = 5.54f;                 // предложение 3
long baseArea = width * length;      // предложение 4
double volume = baseArea * height;   // предложение 5
```

**Рис. 3.14.** Использование разнотипных данных в арифметических выражениях

Предложения 1–3 объявляют переменные для представления ширины (имя `width`, тип `short`), длины (имя `length`, тип `int`) и высоты (имя `heigh`, тип `float`). В левой части предложения 4 объявляется переменная для представления площади основания (имя `baseArea`, тип `long`). В правой части предложения 4 записано арифметическое выражение, вычисляющее площадь основания и использующее разнотипные переменные. В итоге это выражение возвращает значение типа `int`, поскольку это наиболее «длинный» тип данных, использованных в выражении. Операция присваивания в предложении 4 допустима, поскольку данное типа `int` присваивается в переменную типа `long`. Арифметическое выражение в правой части предложения 5 возвращает значение типа `float` (тип переменной `height`). Операция присваивания в предложении 5 также допустима, поскольку данное типа `float` может быть записано в переменную более «длинного» типа `double` без потери информации.

Любое арифметическое выражение, включающее арифметические операции, предполагает несколько логических возможностей последовательности выполнения операций, задаваемых этим выражением. Например, выражение

```
5 + 4 * 3
```

задаёт операции сложения и умножения. Вначале можно выполнить сложение, а затем умножение и, в результате, получить 27. Либо вначале выполнить умножение, а затем сложение и, в результате, получить 17. Поэтому необходимы правила, устраняющие неопределённость в подобных случаях и задающие определённый порядок выполнения операций.

Лучшим правилом является использование скобочной формы записи. Если мы хотим, чтобы вначале выполнилось сложение, а затем умножение, то необходимо записать

```
(5 + 4) * 3
```

Если же мы хотим, чтобы вначале выполнилось умножение, а потом сложение, то необходимо записать

```
5 + (4 * 3)
```

Если придерживаться этого правила, то арифметические выражения всегда однозначны и понятны. Однако, анализируя чужой код, можно встретить выражения, в которых отсутствуют скобки. Это означает, что автор выражения пользуется принятыми в Java правилами, определяющими последовательность выполнения арифметических и логических операций, задаваемых выражениями. Эти правила основаны на *приоритетах операций*. Чем выше приоритет, тем раньше выполняется операция.

В таблице на рис. 3.15 приведены операторы арифметических и логических операций, расположенные в порядке убывания приоритетов. В верхней строке записаны операции с наивысшим приоритетом. Таблицей можно пользоваться как справочником для определения последовательности выполнения операций сложного бесскобочного выражения. Поэтому в ней представлены все операции, включая те, которые будут изучаться в последующих параграфах.

| Символ оператора | Наименование операции |
|---|---|
| `++, --` | инкремент, декремент |
| `*, /, %` | умножение, деление, нахождение остатка от деления |
| `+, -` | сложение/конкатенация, вычитание |
| `<, >, <=, >=` | меньше/больше чем, меньше/больше чем или равно |
| `==, !=` | эквивалентно, не эквивалентно |
| `=, +=, -=, /=, *=, %=` | присваивание, операция и присваивание |

**Рис. 3.15.** Арифметические и логические операции в порядке уменьшения приоритетов

Таким образом, бесскобочное выражение

```
a + b * c
```

предполагает, что вначале выполняется операция умножения, а затем — операция сложения.

### 3.1.3. Составные операторы присваивания. Операторы инкремента и декремента

В практике императивного программирования часто необходимо модифицировать некоторую переменную при помощи арифметической операции, а затем заменить предыдущее значение переменной на новое, полученное в результате модификации. Подобная модификация переменной может осуществляться при помощи предложения с оператором присваивания, у которого в левой и правой частях записывается имя модифицируемой переменной. На рис. 3.16 приведено несколько предложений, в которых переменная `value` модифицируется отмеченным способом.

```
value = value + something;
value = value * something;
value = value + 10.5;
value = value * 1.5;
```

**Рис. 3.16.** Модификация переменной с использованием обычных операторов присваивания

Предложения, похожие на те, которые приведены на рис. 3.16, встречаются настолько часто, что в Java введены специальные *составные операторы присваивания*, упрощающие их запись. При помощи составных операторов присваивания, записанные выше предложения можно переписать в виде, приведенном на рис. 3.17.

```
value += something;
value *= something;
value += 10.5;
value *= 1.5;
```

**Рис. 3.17.** Модификация переменной с использованием составных операторов присваивания

В левой части составного оператора присваивания всегда записывается имя переменной, а сам оператор означает, что вначале необходимо выполнить арифметическую операцию, предусмотренную составным оператором, над левой и правой частями, рассматривая их в качестве операндов, а затем записать полученный результат в переменную левой части. Каждая арифметическая операция, приведенная в таблице на рис. 3.7, может использоваться в соответствующем составном операторе присваивания.

Использование составного оператора присваивания в коде более предпочтительно, поскольку уменьшает вероятность ошибки при повторной записи имени переменной в правой частях. Эта вероятность тем выше, чем длиннее имя переменной.

Частным случаем составных операторов присваивания являются операторы, увеличивающие или уменьшающие значение переменной целочисленного типа на

единицу. Увеличение или уменьшение переменной на единицу можно записать с использованием составных операторов присваивания следующим образом.

```
value += 1;    value -= 1;
```

Однако, поскольку эти операции используются часто, в языке программирования Java введены специальные операторы *инкремента* и *декремента*, увеличивающие или уменьшающие значение целочисленной переменной на единицу.

```
++value;   // оператор инкремента
--value;   // оператор декремента
```

Операторы инкремента и декремента применяются *только для модификации значения целочисленной переменной*. Их нельзя применять для модификации переменной, предназначенной для представления чисел с плавающей запятой, константы или арифметического выражения. Однако, операторы инкремента и декремента могут использоваться внутри арифметического выражения. Приведенные ниже предложения являются недопустимыми.

```
++10;          ++(numb1 + numb2);
```

Опрераторы инкремента и декремента имеют две формы записи: префиксная и постфиксная.

```
++value; // префиксная форма оператора инкремента
value++; // постфиксная форма оператора инкремента
--value; // префиксная форма оператора декремента
value--; // постфиксная форма оператора декремента
```

При использовании префиксной формы записи удвоенный символ операции сложения или вычитания записывается перед именем переменной, а при использовании постфиксной — после имени переменной. В простых предложениях, например, в тех, которые приведены выше, результат выполнения операторов, записанных в префиксной или постфиксной формах, будет один и тот же. Однако, когда переменная, модифицируемая операторами инкремента или декремента находится в правой части предложения с оператором присваивания, то ситуация меняется, и форма записи влияет на результат. Различие иллюстрируется примером кода, приведенным на рис. 3.18.

```
int x = 12;
int y1,y2;
   y1 = ++x; // в итоге y1 равен 13
   y2 = x++; // в итоге y2 равен 12
```

**Рис. 3.18.** Различие в «работе» префиксной и постфиксной формах оператора инкремента

Анализируя код, приведенный на рис. 3.18, можно легко догадаться, как «работают» префиксная и постфиксная формы записи. При префиксной форме записи переменная вначале подвергается модификации, а затем выполняется присваивание. При постфиксной форме записи вначале выполняется присваивание, а затем — модификация переменной.

## 3.2. Булевы данные и выражения

Булевы переменные необходимы для представления булевых литералов и значений булевых выражений. Имеется только два булевых литерала: `true` (истина) и `false` (ложь) и, следовательно, любая булева переменная либо булево выражение могут принимать одно из этих двух значений. Для обозначения того факта, что некоторое поле класса или локальная переменная метода предназначены для хранения данного булевого типа, используется служебное слово

```
boolean
```

Для хранения данных типа `boolean` в памяти компьютера выделяется один байт. Слово «булево» произошло от фамилии английского математика Джорджа Буля, предложившего в конце 19 столетия алгебру, в которой переменные могли принимать только одно из двух значений.

Как и в случае переменных числовых типов, прежде чем использовать булеву переменную в коде, ее необходимо объявить. На рис. 3.19 приведены примеры объявления булевых переменных.

```
boolean q;                // предложение 1

boolean q = true;         // предложение 2

boolean cond1,            // предложение 3
        cond2;

boolean cond1 = true,     // предложение 4
        cond2 = false;
```

**Рис. 3.19.** Объявление булевых переменных

Предложение 1 только объявляет булеву переменную с именем `q`, но не присваивает ей никакого значения. Предложение 2 объявляет переменную `q` и присваивает ей значение, равное `true`. Справа от символа операции присваивания может быть записано логическое выражение и/или вызов метода, возвращающего значение

булевого типа. Предложение 3 объявляет список переменных (`cond1` и `cond2`), а предложение 4 иллюстрирует совмещение объявления списка переменных с заданием им начальных значений.

В практике императивного программирования часто используются булевы выражения. Булевы выражения могут иметь сложную структуру, но их возвращаемое значение всегда есть `true` или `false`.

Булево выражение иногда называют логическим выражением, которое, однако, не следует отождествлять с логическим предложением, как оно понимается в математической логике. Если, например, при записи логических предложений в рамках пропозициональной логики допустимы только пропозициональные символы и логические связки/операторы (отрицание, конъюнкция, дизъюнкция и др.), то при записи булевых выражений используется гораздо более широкий спектр операций.

Простейшим видом булевого выражения является литерал булевого типа, либо имя ранее объявленной (см. рис. 3.19) булевой переменной. Таким образом, любое из слов

```
true    false   cond1  cond2
```

является булевым выражением.

Более сложным и часто используемым видом булевого выражения является выражение, в котором основной операцией является операция сравнения. Данные, которые подвергаются сравнению, записываются слева и справа от символа операции сравнения. Сравниваться могут либо значения переменных, либо возвращаемые значения арифметических выражений. В таблице на рис. 3.20 приведены операции сравнения, используемые при записи булевых выражений сравнения.

| **Символ оператора сравнения** | **Наименование операции сравнения** |
|---|---|
| < | меньше чем |
| <= | меньше чем или равно |
| > | больше чем |
| >= | больше чем или равно |
| != | не равно (не эквивалентно) |
| == | равно (эквивалентно) |

**Рис. 3.20.** Операции сравнения, используемые при записи булевых выражений

На рис. 3.21 приведены примеры булевых выражений сравнения. В левой части выражения (1) записана переменная `numbOfStudents` (количество студентов), а в правой — целочисленный литерал `24`. Это выражение принимает значение `true`,

когда значение переменной numbOfStudents, больше, чем число 24, и значение false, когда значение переменной numbOfStudents меньше либо равно числу 24.

```
numbOfStudents > 24                                          (1)
numbOfStudents <= minNumber                                  (2)
price*numbOfItems != 50.0                                    (3)
(price*numbOfItems) != 50.0                                  (4)
numbOfMen + numbOfWomen == totalNumb – numbOfChildren        (5)
(numbOfMen + numbOfWomen) == (totalNumb – numbOfChildren)    (6)
```

**Рис. 3.21.** Примеры булевых выражений сравнения

В левой части выражения (2) записана переменная numbOfStudents, а в правой части — переменная minNumber (минимальное количество). Выражение принимает значение true, когда количество студентов, представляемое значением переменной numbOfStudents, меньше либо равно значению переменной minNumber и значение false, когда количество студентов, задаваемое значением переменной numbOfStudents, строго больше значения переменной minNumber.

В левой части выражения (3) записано арифметическое выражение price*numbOfItems, представляющее собой произведение значений двух переменных price (цена) и numbOfItems (количество единиц товара), а в правой — литерал в форме с плавающей запятой 50.0. Выражение принимает значение true только в том случае, когда возвращаемое значение отмеченного арифметического выражения не равно числу 50.0, и значение false в том случае, когда возвращаемое значение точно равно числу 50.0. Запись выражения (3) выполнена в соответствии с принятыми приоритетами выполнения операций, приведенными в таблице на рис. 3.15. Выражение (4) представляет собой скобочную форму записи выражения (3).

В левой части выражения (5) записано арифметическое выражение numbOfMen+numbOfWomen, представляющее собой сумму значений переменных numbOfMen (количество присутствующих мужчин) и numbOfWomen (количество присутствующих женщин), а в правой — арифметическое выражение totalNumb - numbOfChildren, представляющее собой разность значений переменных totalNumb (общее количество присутствующих) и numbOfChildren (количество присутствующих детей). Выражение (5) принимает значение true только в том случае, когда количество присутствующих мужчин и женщин в точности равно разности между общим количеством присутствующих и количеством присутствующих детей, и значение false, когда количество присутствующих мужчин и женщин не равно разности между общим количеством присутствующим и количеством присутствующих детей. Выражение (6) представляет собой скобочную форму записи выражения (5).

Еще одним видом булевого выражения является выражение, составленное из булевых переменных либо выражений и логических операций над ними. В таблице на рис. 3.22 приведены основные логические операции, используемые при записи булевых выражений.

| Символ логического оператора | Наименование логической операции |
|---|---|
| ! | отрицание |
| && | логическое «И» |
| \|\| | логическое «ИЛИ» |
| ^ | исключающее «ИЛИ» |

**Рис. 3.22.** Логические операции, используемые при записи булевых выражений

Операторы «И» и «ИЛИ» являются бинарными, и их действие такое же, как и действие логических связок «конъюнкция» и «дизъюнкция» в пропозициональной логике, соответственно. Оператор «исключающее ИЛИ» также является бинарным. Он возвращает значение `true` в том случае, когда оба операнда различны. Если оба операнда имеют одинаковое значение, то операнд возвращает значение `false`.

Оператор «отрицание» является унарным и применяется к одному операнду. Это может быть либо одна булева переменная, либо один булев литерал, либо одно булево выражение. Результатом выполнения операции «отрицание» является изменение значения операнда на противоположное. Если, например, значение некоторой переменной до применения к ней операции отрицания есть `true`, то после применения операции отрицания оно становится равным `false`. На рис. 3.23 приведены булевы выражения, которые иллюстрируют случаи применения операции отрицания.

```
!true                          (1)
!alarmOn                       (2)
!(numbOfStudents > 24)         (3)
```

**Рис. 3.23.** Случаи применения операции логического отрицания

Операции логического «И» и логического «ИЛИ» являются бинарными и выполняются над двумя операндами, в качестве которых могут фигурировать булевы переменные или выражения.

Результатом выполнения операции «И» является значение `true` только в том случае, когда значения обоих операндов также равны `true`. Для всех остальных возможных комбинаций значений операндов (`true-false`, `false-true`, `false-false`) результатом выполнения операции «И» является значение `false`.

Результатом выполнения операции «ИЛИ» является значение `false` только в том случае, когда значения обоих операндов равны `false`. Для всех остальных возможных комбинаций значений операндов результатом выполнения операции «ИЛИ» является значение `true`.

На рис. 3.24 приведены булевы выражения, которые иллюстрируют применение всех трех логических операций.

```
alarmOn && timeToAwake                          (1)
alarmOn && (!timeToSleep)                       (2)
christmas || newYear                            (3)
(christmas && sale) || (newYear && sale)        (4)
```

**Рис. 3.24.** Использование логических операций в булевых выражениях

На рис. 3.24 выражение (1) принимает значение `true` только в том случае, когда обе булевы переменные `alarmOn` (будильник включен) и `timeToAwake` (время просыпаться) также принимают значение `true`. Если хотя бы одна из переменных `alarmOn` или `timeToAwake` принимает значение `false`, то выражение (1) также принимает значение `false`.

Выражение (2) принимает значение `true` в том случае, когда переменная `alarmOn` принимает значение `true`, а переменная `timeToSleep` (время спать) — значение `false`. При всех остальных комбинациях значений переменных `alarmOn` и `timeToSleep` выражение (2) принимает значение `false`.

Выражение (3) принимает значение `true`, когда хотя бы одна из переменных `christmas` (Рождество) либо `newYear` (новый год) приняла значение `true`. Для того, чтобы выражение (3) приняло значение `false`, необходимо, чтобы обе переменные `christmas` и `newYear` приняли значение `false`.

Выражение (4) принимает значение `true` в тех случаях, когда хотя бы одно из выражений-операндов `(christmas && sale)` либо `(newYear && sale)` принимает значение `true`. Несложно проверить, что приведенные ниже комбинации значений переменных `christmas`, `newYear` и `sale` (распродажа) определяют значение `true` для всего выражения (4).

Комбинация 1 (`christmas = true`, `newYear = false`, `sale = true`)
Комбинация 2 (`christmas = false`, `newYear = true`, `sale = true`)
Комбинация 3 (`christmas = true`, `newYear = true`, `sale = true`)

Наиболее универсальным видом булевого выражения является выражение, в котором комбинируются операции сравнения, приведенные в таблице на рис. 3.20, и логические операции, приведенные в таблице на рис. 3.22. В таких выражениях часто операндами логических операций «И», либо «ИЛИ» фигурируют выражения сравнения. Для облегчения понимания смысла булевских выражения этого вида порядок выполнения операций желательно задавать явно, при помощи скобочной формы записи. На рис. 3.25 приведено несколько примеров таких булевых выражений.

```
(numbOfStudents >= 20) && (numbOfStudents <= 30)          (1)
(numbOfStudents < 20) || (numbOfStudents > 30)            (2)
```

**Рис. 3.25.** Комбинация логических операций и операций сравнения в булевых выражениях

Выражение (1) принимает значение `true` только в том случае, когда количество студентов, определяемое значением переменной `numbOfStudents`, находится внутри сегмента [20, 30], а выражение (2) — во всех случаях, когда количество студентов выходит за пределы этого сегмента.

## 3.3. Тип данных для представления знаков и символов

Для представления одного из символов, нанесенных на клавиатуру компьютера, а также других знаков и символов используются переменные примитивного типа, обозначаемого служебным словом

```
char
```

В языках программирования предыдущих поколений переменная типа `char` использовалась, главным образом, для представления символов клавиатуры и некоторых других символов, не нанесенных на клавиатуру, например, символов, позволяющих рисовать рамки и таблицы: « ┌ », « ╤ ». Для представления такого рода символов было достаточно одного байта. В языке программирования Java для хранения данных типа `char` выделяется два байта, что в сочетании с методом кодирования знаков, получившим наименование Unicode, позволяет представить *более одного миллиона различных знаков*. Этого достаточно, чтобы представить все символы, которыми люди пользовались в прошлом и используют сейчас для записи текстов, плюс большое количество специальных знаков, не относящихся к письму. При этом еще остается значительное количество зарезервированных кодовых комбинаций, которые могут понадобиться в будущем. Метод кодирования Unicode позволяет, наряду с английским, использовать национальные языки в различных сферах современных информационных технологий.

Таким образом, литералами типа `char` могут быть любые знаки, представимые в Unicode. Существуют два способа записи литералов типа `char`. Первый способ весьма прост, но ограничен тем, что позволяет представлять только литералы, нанесенные на клавиатуру компьютера. При использовании этого способа литерал — это один из символов клавиатуры, заключенный в апострофы. Рис. 3.26 иллюстрирует первый способ записи литералов типа `char`.

```
char rusLetter = 'Ы',
     engLetter = 'S',
     numeral   = '9',
     symbol    = '+';
```

**Рис. 3.26.** Примеры записи литералов типа `char`, нанесенных на клавиатуру компьютера

Второй способ записи литералов несколько сложнее, но является универсальным и позволяет представлять любой из знаков, охватываемых системой Unicode. При использовании этого способа в апострофы заключается Unicode-номер литерала, предваряемый префиксом «\u». Unicode-номер литерала представляет собой четырехразрядное шестнадцатеричное число, уникальное для любого знака в системе кодирования Unicode. Рис. 3.27 иллюстрирует способ записи литералов типа char при помощи Unicode-номера.

```
char rusLetter = '\u042B',  // русская буква Ы
     engLetter = '\u0053',  // английская буква S
     numeral   = '\u0039',  // цифра 9
     symbol    = '\u002B';  // символ +
```

**Рис. 3.27.** Примеры записи литералов типа char в виде Unicode-номера

Трудно запомнить Unicode-номера всех знаков, поэтому для записи литералов типа char вторым из отмеченных способов необходим дополнительный источник справочной информации. Существует множество источников информации с таблицами, ставящими в соответствие внешнему виду знака, воспринимаемому нашим зрением, его Unicode-номер. Например, текстовый редактор Word позволяет вставить в редактируемый текст символы, отсутствующие на клавиатуре, при помощи таблицы дополнительных символов. Панель, которая включает эту таблицу, одновременно отображает Unicode-номер каждого выбираемого символа.

Для представления любого разряда шестнадцатеричного числа достаточно четырех бит или тетрады. Поэтому любой Unicode-номер (представимый четырехразрядным шестнадцатеричным числом) размещается в двух байтах, отводимых для представления данных типа char. Рис. 3.28 иллюстрирует внутреннее представление литералов, использованных в предыдущих примерах. Каждый шестнадцатеричный разряд Unicode-номера записан в виде отдельной тетрады.

```
char rusLetter = '\u042B', // внутреннее представление «Ы» 0000 0100 0010 1011
     engLetter = '\u0053', // внутреннее представление «S» 0000 0000 0101 0011
     numeral   = '\u0039', // внутреннее представление «9» 0000 0000 0011 1001
     symbol    = '\u002B'; // внутреннее представление «+» 0000 0000 0010 1011
```

**Рис. 3.28.** Примеры записи литералов типа char в виде Unicode-номера

Как видно из примеров внутреннего представления данных типа char, их можно рассматривать как целочисленные данные типа short, для представления которых также выделяется два байта. И, следовательно, данные типа char можно присваивать в переменные типа short, int и long без потери информации (см. таблицу на рис. 3.1). В том случае, когда мы уверены, что присваивание

данных типа char в переменную типа byte не приводит к потере информации, это также можно сделать при условии использования оператора явного преобразования типа.

Рис. 3.29 иллюстрирует варианты присваивания значений переменной типа char в переменные, имеющие один из целочисленных типов.

```
char symbol = '+';
byte a;
int b;
long c;
   b = symbol;
   c = symbol;
   a = (byte)symbol;
```

**Рис. 3.29.** Присваивание значений переменной типа char переменным целочисленных типов

### 3.3.1. Булевы выражения с данными типа char

Поскольку данные типа char могут рассматриваться как целочисленные данные, то их можно сравнивать между собой и использовать в булевых выражений. Два символа будем считать равными, если они имеют одинаковые Unicode-номера, а из двух символов большим будет тот, у которого больше значение Unicode-номера. Код на рис. 3.30 иллюстрирует построение булевских выражений на основе операции сравнения значений переменных типа char.

```
char letterA = 'a';  // Unicode-номер \u0061
char letterB = 'b';  // Unicode-номер \u0062

letterA == letterB   (1)
letterA > letterB    (2)
letterA < letterB    (3)
```

**Рис. 3.30.** Сравнение данных типа char

На рис.3.30 объявлены две переменные letterA и letterB типа char, которым присвоены значения английских букв 'a' и 'b' соответственно. Затем составлены три булевых выражения с использованием операции сравнения значений переменных letterA и letterB.

Выражение (1) принимает значение false, потому что Unicode-номера символов 'a' и 'b' не совпадают. Поскольку Unicode-номер символа 'a' меньше Unicode-номера символа 'b', то выражение (2) принимает значение false, а выражение (3) — значение true.

Булевы выражения с данными типа `char` могут строиться путем комбинации операций сравнения и логических операций. Несколько примеров таких булевых выражений приведено на рис. 3.31.

```
char firstLetter = 'a';    // Unicode-номер \u0061
char secondLetter = 'b';   // Unicode-номер \u0062
char thirdLetter = 'c';    // Unicode-номер \u0063

(firstLetter < secondLetter) && (secondLetter < thirdLetter)    (1)
(firstLetter < secondLetter) || (thirdLetter < secondLetter)    (2)
```

**Рис. 3.31.** Булевы выражения с данными типа `char`

На рис. 3.31 булево выражение (1) составлено из двух выражений сравнения, связанных логической операцией «И», а выражение (2) — из двух выражений сравнения, связанных логической операцией «ИЛИ». Очевидно, что оба булевых выражения принимают значение `true`.

Упорядочение Unicode-номеров для букв многих европейских национальных языков соответствует расположению этих букв в национальных алфавитах. Поэтому словосочетание «буква Y больше буквы X» можно понимать как «буква Y расположена в алфавите после буквы X». Это обстоятельство позволяет разрабатывать программы сортировки наборов символов, упорядочивая их расположение в соответствии с алфавитом.

В булевых выражениях могут использоваться данные типа `char`, значения которых не нанесены на клавиатуру. В этом случае их значения записываются в виде Unicode-номеров.

### 3.3.2. Организация пространства знаков в Unicode

Одной из отличительных особенностей языка программирования Java является широкое использование системы Unicode для представления знаков. Некоторые радикально настроенные авторы даже утверждают, что если бы не было Unicode, то не было бы и Java. Язык программирования Java позволяет использовать Unicode не только для представления отдельных знаков и строк знаков в виде литералов, но и для записи имен переменных и других идентификаторов кода. Последняя возможность редко применяется на практике, поскольку хороший стиль программирования предполагает использование английского языка как для записи служебных слов, так и для записи идентификаторов. При удачном подборе имен переменных предложение на Java приближается к англоязычному предложению и становится более понятным. Тем не менее, в Интернет встречаются примеры исходного кода на Java с использованием букв греческого алфавита и национальных языков при записи имен переменных и других идентификаторов.

Из содержания предыдущего параграфа могло сложиться впечатление, что знак в Unicode представляется только в виде четырехразрядного шестнадцатеричного числа. Однако, это только одна из характеристик представления знака в Unicode. Более полный список характеристик, идентифицирующих знак в Unicode, включает следующие характеристики.

- Визуальный образ знака в виде одной конкретной, но типичной (репрезентативной) визуальной формы, которую может принимать знак.
- Уникальный Unicode-номер, который никогда не изменяет своего значения.
- Уникальное Unicode-имя, которое также никогда не изменяет своего значения, даже если обнаруживается, что оно ошибочно по сути либо было записано с грамматической ошибкой. Unicode-имя является, скорее, мнемоническим идентификатором знака, чем именем в обычном значении этого слова.
- Множество свойств, представленное в формализованной форме. Свойства описывают, например, принадлежность знака к одной из групп знаков (буквы, цифры, знаки пунктуации и т. п.) и другие свойства.
- Комментарий в виде описаний, которые могут пояснять значение знака, сравнивать данный знак с другими знаками, описывать возможные варианты его визуального восприятия и т. д.

Внутренняя организация множества знаков, охватываемых системой Unicode, изменялась по мере развития Unicode. Современная организация пространства знаков, охватываемых системой Unicode, размещает знаки в кодовых плоскостях.

Каждая кодовая плоскость имеет матричную структуру и состоит из 256 столбцов и 256 строк. Таким образом, кодовая плоскость может представлять 65536 знаков. Согласно общепринятой договоренности, количество кодовых плоскостей, которые будут заполняться по мере включения в систему Unicode новых знаков, никогда не будет превышать 17. Таким образом, пространство знаков Unicode может вмещать 1114112 знаков.

Рис. 3.32 иллюстрирует структуру пространства знаков Unicode в виде набора из 17 кодовых плоскостей.

Кодовые плоскости на рис. 3.32 пронумерованы шестнадцатеричными цифрами. Как видно на рис. 3.32, большая часть пространства знаков Unicode еще не заполнена и зарезервирована для развития системы. Свободными остаются 11 плоскостей с номерами 3–D (шестнадцатеричные).

Плоскость с номером 0 (шестнадцатеричное) называется «Базовая многоязыковая плоскость» и в англоязычной литературе обозначается аббревиатурой BMP (Basic Multilingual Plane). Эта плоскость включает практически все знаки, используемые в приложениях, ориентированных на систему Unicode. Знаков, представленных в BMP, достаточно для представления текстов любой сложности на любом из современных языков.

Плоскость с номером 1 (шестнадцатеричное) называется «Дополнительная многоязыковая плоскость» и в англоязычной литературе обозначается аббревиатурой SMP (Supplementary Multilingual Plane). Эта плоскость предназначена для представления знаков древних, архаических, а также современных, но вышедших из


**Рис. 3.32.** Структура пространства знаков системы Unicode в виде набора из 17 кодовых плоскостей

употребления систем письменности. Главными потребителями такого рода знаков являются ученые, изучающие древние и старинные языки и тексты. Эта плоскость содержит также наборы различных и специализированных знаков.

Плоскость с номером 2 (шестнадцатеричное) носит наименование «Дополнительная идеографическая плоскость» и в англоязычной литературе обозначается

аббревиатурой SIP (Supplementary Ideographic Plane). Эта плоскость представляет собой расширение области для представления идеографических знаков, которая имеется в BMP, и содержит редкие и необычные знаки Китайского идеографического письма. Идеография — это принцип письма, при котором единицей графического обозначения является не буква, а слово или морфема.

Плоскость с номером E (шестнадцатеричное) называется «Дополнительная плоскость специального назначения» и в англоязычной литературе обозначается аббревиатурой SSP (Supplementary Special-Purpose Plane). Панель зарезервирована для хранения кодов, при помощи которых кодируются не знаки, а команды протоколов высокого уровня, используемых для обработки Unicode-текстов.

Плоскости с номерами F и 10 (шестнадцатеричные) называются «Плоскости приватного использования», не имеют специальной англоязычной аббревиатуры и представляют собой расширение области для приватного использования, которая имеется в BMP.

Для записи кода на языке программирования Java важна кодовая плоскость номер 0, или базовая многоязычная плоскость BMP. Двух байт, которые выделяются в языке программирования Java для представления встроенных данных типа `char`, достаточно для записи Unicode-номера любого знака этой плоскости, а литерал типа `char` в языке программирования Java — это, по сути, один из знаков кодовой плоскости BMP. Отметим, что для записи Unicode-номеров знаков из кодовых плоскостей с номерами 1–10 (шестнадцатеричные) необходимо использовать специальный способ кодирования, именуемый механизмом замещения (surrogate mechanism).

## 3.4. Классы-оболочки для примитивных типов данных

Каждому примитивному типу данных в языке программирования Java соответствует одноименный предопределенный класс, объекты которого могут использоваться для представления соответствующих данных. Эти предопределенные классы называются классами-оболочками примитивных типов данных. Таким образом, в языке программирования Java имеются следующие классы-оболочки.

```
Byte, Short, Integer, Long, Float, Double, Boolean, Character
```

Представление числовых, булевых и символьных данных в виде объектов классов-оболочек превращает имя данного в ссылочную переменную и дает более полную информацию о данных, чем их представление в виде примитивных данных. В практике императивного программирования представление числовых, булевых и символьных данных в виде объектов часто является излишним, и для написания кода достаточно имеющихся примитивных данных. Однако, в некоторых случаях, нельзя обойтись без классов-оболочек.

Предопределенные классы-оболочки примитивных данных обладают следующими отличительными особенностями, которые детерминируют сферу их применимости.

- Позволяют рассматривать числовые, булевы и символьные данные как объекты, создавать эти объекты и ссылки на них.
- Предоставляют поля, содержащие информацию о литералах данного соответствующего типа. Например, значения литералов для булевых данных, наибольшие и наименьшие значения литералов для числовых данных и т. п.
- Предоставляют методы для выполнения часто используемых операций над данными соответствующего типа. Например, методы, осуществляющие преобразование строки символов в числовое данное соответствующего типа.

Прежде чем использовать объект одного из классов-оболочек, он должен быть создан. Создание объектов какого либо класса, в том числе и класса-оболочки, осуществляется предложением со служебным словом `new`. В этом предложении должны быть специфицированы:

(1) имя и тип ссылочной переменной на вновь созданный объект;
(2) имя метода-конструктора, который участвует в начальной инициализации полей объекта;
(3) фактические значения параметров конструктора, если таковые имеются.

Рис. 3.33 иллюстрирует структуру предложения, при помощи которого создается объект класса `Integer` , которому при начальной инициализации присваивается значение 55.

В левой части предложения, приведенного на рис. 3.33, объявлена переменная ссылочного типа, хранящая ссылку на вновь созданный объект и являющаяся именем созданного объекта.

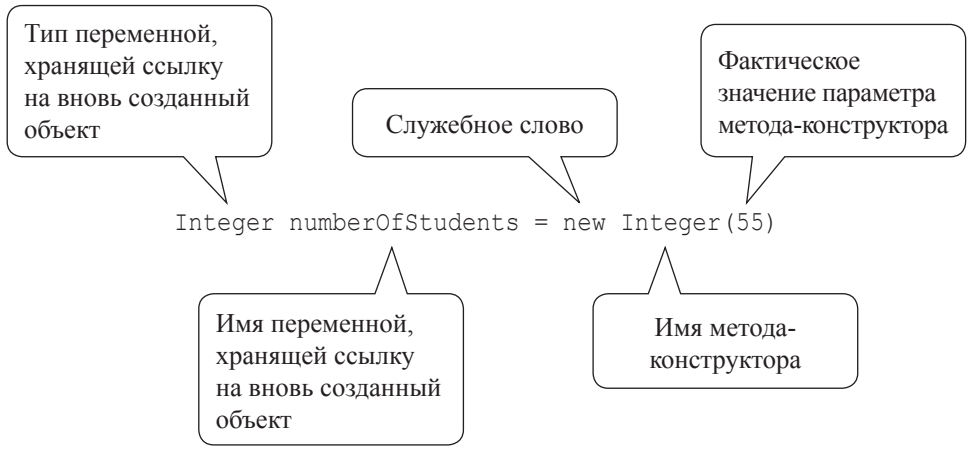



**Рис. 3.33.** Структура предложения, при помощи которого создаются новые объекты классов-оболочек

Типом этой ссылки является имя класса, используемого при создании объекта. В правой части предложения после служебного слова `new` указывается метод-конструктор, который используется для задания начальных значений полей при их начальной инициализации. Имя конструктора всегда совпадает с именем класса, используемого при создании объекта. Если конструктор имеет входные параметры, то их фактические значения перечисляются в скобках после имени конструктора.

Для более подробного изучения классов-оболочек необходимы дополнительные знания о классах и интерфейсах, поэтому мы вернемся к классам-оболочкам в последующих частях книги.

## Упражнения для самостоятельной работы

3.1. Разработайте программный код для вычисления наибольшего десятичного числа, которое может быть записано в однобайтный двоичный регистр.

3.2. Разработайте программный код для вычисления площади монеты, имеющей отверстие в центре.

3.3. Используя код, полученный при выполнении упражнения 3.2, разработайте программу, вычисляющую объём и вес монеты.

3.4. Разработайте программный код для вычисления количества рулонов обоев, необходимых для оклеивания стены. Считайте, что стена не имеет дверей и окон.

3.5. Используя код, полученный при выполнении упражнения 3.4, разработайте программу, вычисляющую количество рулонов обоев, необходимых для оклеивания стены, имеющей произвольное количество дверей и окон.

3.6. Используя код, полученный при выполнении упражнения 3.5, разработайте программу, вычисляющую количество рулонов обоев, необходимых для оклеивания комнаты. Считайте, что стены могут иметь произвольное количество дверей и окон.

3.7. Врач назначил больному лекарственный препарат в виде таблеток. Разработайте программный код, вычисляющий количество упаковок лекарственного препарата, которые должен купить больной, и общую стоимость лекарства, если известно:
- количество таблеток, которые больной должен принимать за один раз;
- количество приёмов лекарства в течение дня;
- длительность лечения в днях;

— количество таблеток в упаковке;
— стоимость одной упаковки.

3.8. Разработайте программный код для вычисления среднего количества студентов, присутствующих на поточных лекциях, если известно, что: (1) поток состоит из четырех групп; (2) в первой группе учится 24 студента, из которых, в среднем, 10% студентов пропускают занятия; (3) во второй группе учится 22 студента, из которых, в среднем, 12% пропускают занятия; (4) в третьей группе учится 20 студентов, из которых, в среднем, 9% пропускают занятия; (5) в четвертой группе учится 25 студентов, из которых, в среднем, 17% пропускают занятия.

3.9. Разработайте программный код для нахождения корней квадратного уравнения в том случае, когда значение дискриминанта квадратного уравнения больше нуля.

3.10. Разработайте программный код для перевода значения температуры из шкалы Фаренгейта в шкалу Цельсия. Если значение температуры представлено в шкале Фаренгейта, то для перевода этого значения в шкалу Цельсия необходимо от исходного числа отнять 32 и умножить результат на 5/9.

3.11. Разработайте программный код для перевода значения температуры из шкалы Цельсия в шкалу Фаренгейта. Если значение температуры представлено в шкале Цельсия, то для перевода этого значения в шкалу Фаренгейта необходимо умножить исходное число на 5/9 и к результату прибавить 32.

3.12. Разработайте программный код для вычисления примерного веса атмосферы Земли. При кодировании арифметических выражений не используйте более одной арифметической операции в одном выражении. Столб земной атмосферы давит на один квадратный сантиметр Земной поверхности с силой 1 кг. Средний радиус Земли равен примерно 6 371 км.

3.13. Разработайте программный код для вычисления полного количества часов и минут, прошедших от начала суток и до того момента (в первой половине дня), когда часовая стрелка повернулась на угол $\varphi$ градусов ($0 \le \varphi < 360°$)/

3.14. Разработайте программный код для вычисления угла (в градусах) между положением часовой стрелки в начале суток и ее положением в текущее время.

Раздел 4

# СТРОКОВЫЕ ДАННЫЕ И ОПЕРАЦИИ СО СТРОКАМИ

Строковые данные — это последовательности, представленные знаками или символами типа `char`. Чаще всего строковые данные используются для представления слов и фраз естественного языка. Использование системы Unicode позволяет относительно легко представлять в качестве строковых данных слова и фразы национальных языков и строки, включающие специальные символы, не нанесенные на клавиатуру компьютера.

Длина строки символов может варьироваться в широких пределах, поэтому сложно представлять строковые данные в виде встроенных типов, имеющих фиксированный размер. Для представления строковых данных в язык программирования Java включен предопределенный класс `String`, объектами которого являются строки символов. Таким образом, в Java имеется только один тип для представления строковых данных с именем `String`.

## 4.1. Создание объектов класса `String`

Поскольку строковые данные являются объектами, то, прежде чем их использовать в программном коде, они должны быть созданы. Объекты класса `String` могут создаваться при помощи предложения со служебным словом `new`. Например,

```
String bestCity = new String("Одесса");
```

При помощи этого предложения создается новый объект с именем `bestCity` типа `String` и начальным значением `"Одесса"`. Здесь переменная `bestCity`, имя которой мы можем использовать как имя строки, по сути, является ссылочной переменной типа `String`, хранящей ссылку на сам объект.

Для создания объектов класса `String` можно использовать более простое предложение, похожее на предложения, при помощи которых объявляются переменные встроенных типов. Приведенное выше предложение эквивалентно следующему.

```
String bestCity = "Одесса";
```

Вместо последнего предложения можно записать два предложения, разделив объявление имени строки и присваивание ей начального значения.

```
String bestCity;
bestCity = "Одесса";
```

Первое предложение объявляет переменную с именем `bestCity` и типом `String`, но не создает объект класса `String`. Объект будет создан в результате выполнения второго предложения. Объекту будет присвоено начальное значение `"Одесса"`, а в переменную `bestCity` будет записана ссылка на вновь созданный объект.

Существует два способа записи литералов типа `String`, аналогичные способам записи литералов типа `char`. Как видно из приведенных примеров, строковые литералы могут записываться в виде последовательности символов, нанесенных на клавиатуру компьютера, заключенные в двойные апострофы. Более универсальная форма записи предполагает использование Unicode-номера символа или знака. Рис. 4.1 иллюстрирует возможные варианты записи строковых литералов.

```
String bestCity = "Одесса";                                    // предложение 1
String bestCity = "\u041E\u0434\u0435\u0441\u0441\u0430"; // предложение 2
String bestCity = "Одесс\u0430";                               // предложение 3
```

**Рис. 4.1.** Способы записи литералов типа `String`

Во всех трех предложениях на рис. 4.1 строке с именем `bestCity` присваивается одно и то же начальное значение. В первом предложении строка состоит из последовательности символов, нанесенных на клавиатуру компьютера. Во втором предложении эти же символы заменены их Unicode-номерами. В третьем предложении Unicode-номер используется только для представления последнего символа.

## 4.2. Конкатенация строк

Одной из самых распространенных операций над строковыми данными является операция сцепления строк, называемая также «операция конкатенации». В результате выполнения операции конкатенации из нескольких объектов типа `String` создается новый объект типа `String`. Новый объект представляет собой строку символов, полученную путем соединения/сцепления строк символов, участвующих в операции конкатенации. Рис. 4.2 иллюстрирует символ оператора конкатенации и его работу.

```
String firstName = "Александр ";
String secondName = "Пушкин";
String fullName = firstName + secondName;
```

**Рис. 4.2.** Конкатенация строк. Операндами являются имена строк

Первые два предложения на рис. 4.2 создают два объекта типа `String`, присваивают им значения `"Александр "` и `"Пушкин"` и объявляют ссылки на эти объекты с именами `firstName` и `secondName`. Третье предложение создает еще один объект типа `String` и формирует его значение путем конкатенации значений двух предыдущих объектов. Переменная с именем `fullName` является ссылкой на созданный объект. Символом операции конкатенации является «+». В результате выполнения третьего предложения значением объекта `fullName` будет строка

```
"Александр Пушкин"
```

В операции конкатенации могут участвовать более двух операндов, а операндами могут быть как имена строк (имена объектов), так и строковые литералы. Рассмотрим код, приведенный на рис. 4.3.

```
String name = "Александр ";          // предложение 1
name = name + "Сергеевич ";          // предложение 2
name = name + "Пушкин";              // предложение 3
```

**Рис. 4.3.** Конкатенация строк. Операндами являются имена строк и строковые литералы

В терминах «имя строки» и «значение строки» предложение 1 на рис. 4.3 интерпретируется следующим образом. Объявляется строка с именем `name` и со значением `"Александр "`. В терминах «класс» и «объект» это же предложение интерпретируется иначе. Создается новый объект класса `String` с начальным значением `"Александр "` и со ссылкой на этот объект, имеющей имя `name`.

В правой части предложения 2 создается новый объект класса `String`. Его значение `"Александр Сергеевич "` формируется путем конкатенации строк `"Александр "` и `"Сергеевич "`. Ссылка на этот новый объект помещается в переменную `name`, заменяя в ней предыдущее значение. Это означает, что после выполнения предложения 2 доступ к объекту со значением `"Александр "` невозможен.

Ясно, что после выполнения предложения 3 формируется еще один объект (третий по счету) класса `String` со значением `"Александр Сергеевич Пушкин"`. Ссылка на этот объект опять помещается в переменную `name`, заменяя в ней предыдущее значение. Теперь становится невозможным доступ к объекту `"Александр Сергеевич "`.

Предложения 2 и 3 на рис. 4.3 осуществляют модификацию строки с применением операции конкатенации, аналогично тому, как осуществляется модификация значения переменной одного из числовых типов при выполнении арифметических

операций. Такой способ модификации строки используется весьма часто, поэтому в языке программирования Java имеется составной оператор присваивания с конкатенацией. С использованием составного оператора присваивания код на рис. 4.3 можно переписать и представить так, как это показано на рис. 4.4.

```
String name = "Александр ";
name += "Сергеевич ";
name += "Пушкин";
```

**Рис. 4.4.** Составной оператор конкатенации и присваивания

Большое количество операций, которые можно выполнять над строковыми данными, осуществляется при помощи методов, объявленных в предопределенном классе `String`.

## 4.3. Методы класса `String`

В классе `String` объявлено около 50 методов, предназначенных для выполнения операций над строковыми данными. Характерной особенностью всех методов, объявленных в классе `String`, является то, что эти методы *не изменяют значение строки*, к которой они применяются. Например, при помощи одного из методов класса `String` можно выделить из строки ее символ, а при помощи другого — часть строки или подстроку, но исходная строка в обоих случаях остается «неповрежденной».

Технология выполнения операций над объектами классов вообще и над объектами класса `String`, в частности, отличается от технологии выполнения операций над примитивными данными.

Кодирование операций над примитивными данными осуществляется путем составления предложений, имеющих, вообще говоря, похожую структуру. Такие предложения, как правило, включают выражение, составленное из имен переменных и литералов, а также символов арифметических либо булевых операций. Результат может быть записан в переменную при помощи оператора присваивания. Такой способ кодирования операций над примитивными типами данными похож на запись выражений, принятых в арифметике, алгебре и математической логике, и поэтому навыки записи арифметических, алгебраических и логических выражений применимы при кодировании операций над примитивными данными.

В классе `String` объявлены методы, выполняющие операции над объектами этого класса. После создания объекта класса `String` он размещается в некотором участке основной памяти компьютера. В этом участке памяти размещается не только значение строки, но и все члены класса `String`, включая его методы. Для того чтобы применить один из методов класса `String` к объекту класса `String` необходимо

этот метод вызвать. Вызов метода объекта осуществляется при помощи сообщения, адресованного объекту. В последующих разделах мы будем неоднократно возвращаться к понятию сообщения и изучать его более подробно и в различных контекстах. Сейчас ограничимся только теми знаниями, которые необходимы для вызова методов класса `String`. Любое сообщение имеет структуру, приведенную на рис. 4.5.

```
<ссылка на объект> . <имя метода>(<список фактических параметров>)
```

**Рис. 4.5.** Структура сообщения для вызова метода

Как видно на рис. 4.5, сообщение состоит из ссылки на объект, в котором размещен вызываемый метод, и заголовка вызываемого метода, разделенные символом «точка».

Ссылкой на объект класса `String` может быть либо имя строки, которое задается при ее создании, либо строковый литерал.

Заголовок метода состоит из имени метода и круглых скобок, содержащих список фактических значений параметров метода, если таковые имеются. Если метод объявлен без параметров, то круглые скобки остаются пустыми.

Таким образом, предложение, при помощи которого кодируется операция над объектом класса `String` путем применения одного из методов этого объекта, должно включать сообщение, вызывающее требуемый метод. Значения, возвращаемые методами класса `String`, часто имеют один из примитивных типов и, следовательно, сообщение может быть включено, в качестве одного из членов, в выражения для выполнения операций над данными с примитивными типами.

Ясно, что для вызова метода класса `String` необходимо иметь ранее созданный объект этого класса и обладать минимально необходимыми знаниями о вызываемом методе. Эти минимально необходимые знания включают: (1) имя метода и его назначение; (2) параметры метода и их смысл; (3) тип возвращаемого методом значения. Рассмотрим некоторые, наиболее употребляемые методы класса `String`, а также примеры, иллюстрирующие их использование.

Многие методы класса `String` используют в качестве входного параметра индекс символа в строке или номер символа в строке. Для правильного применения этих методов необходимо знать принятый способ нумерации символов. Принято, что счет символов в строке ведется слева направо и начинается не с единицы, а с нуля. Таким образом, *первым индексом/номером символа в строке является ноль*.

### 4.3.1. Определение длины строки. Выделение символа или подстроки

Одними из наиболее часто используемых методов класса `String` являются методы `length`, `charAt` и `substring`. На рис. 4.6 приведены заголовки этих методов так, как они объявлены в классе `String`.

```
public int length()
public char charAt(int index)
public String substring(int begin)
public String substring(int begin, int end)
```

**Рис. 4.6.** Заголовки методов `length`, `charAt` и `substring`

Как видно на рис. 4.6, все методы доступны из любой точки программы. Об этом свидетельствует модификатор доступа `public`. Модификаторы доступа языка программирования Java и их возможные значения будут изучаться в последующих разделах, посвященных моделированию и кодированию объектно-ориентированных программ.

При помощи метода `length` можно узнать количество символов в текущей строке. *Текущей строкой будем называть строку-объект, из которой вызывается метод*. Метод `length` не имеет входных параметров и возвращает количество символов в строке в виде целого положительного числа. На рис. 4.7 приведен пример, иллюстрирующий применение метода `length`. Пример построен на основе кода, приведенного ранее на рис. 4.4.

Первое предложение на рис. 4.7 объявляет переменную с именем `numbOfSymbols` (количество символов) типа `int`. При помощи второго предложения создается объект-строка с именем `name` и начальным значением `"Александр "`. В правой части третьего предложения записано сообщение, вызывающее метод `length` объекта `name`. Метод возвращает количество символов этого объекта, равное 10, однако индекс последнего символа строки `name` в предложении 3 будет равен 9, поскольку нумерация символов в строках начинается с нуля. Предложение 4 модифицирует значение строки `name` путем конкатенации предыдущего значения этой строки со строкой `"Сергеевич "` и присваиванием результата в строку `name`. Предложение 5 вычисляет количество символов в модифицированном объекте `name` и обновляет значение переменной `numbOfSymbols`. Предложения 6 и 7 выполняют точно такую же «работу», как и предложения 4 и 5.

```
int numbOfSymbols;              // предложение 1
String name = "Александр ";     // предложение 2
numbOfSymbols = name.length(); // предложенте 3, в numbOfSymbols число 10
name += "Сергеевич ";           // предложение 4
numbOfSymbols = name.length(); // предложение 5, в numbOfSymbols число 20
name += "Пушкин";               // предложение 6
numbOfSymbols = name.length(); // предложение 7, в numbOfSymbols число 26
```

**Рис. 4.7.** Вызов метода `length` с использованием имени строки

В примере на рис. 4.7 вызов метода `length` осуществляется сообщением, у которого в качестве ссылки на строку используется ее имя `name`. Однако, ранее мы отмечали, что в сообщении в качестве ссылки может фигурировать и строковый литерал. Код, приведенный на рис. 4.8, иллюстрирует отмеченную возможность.

```
int numbOfSymbols = "Александр ".length() +
                    "Сергеевич ".length() +
                    "Пушкин".length();       // в numbOfSymbols число 26
```

**Рис. 4.8.** Вызов метода `length` с использованием строкового литерала

В левой части предложения, приведенного на рис. 4.8, объявлена целочисленная переменная `numbOfSymbols`. В правой части предложения, в каждом из трех сообщений, вызывающих метод `length`, в качестве ссылки используется строковый литерал. Поскольку метод `length` возвращает целочисленное значение, то мы можем интерпретировать каждое из сообщений как целочисленное слагаемое в выражении для нахождения суммы трех целых чисел типа `int`. Символ «+» в коде на рис. 4.8 это *символ операции сложения числовых данных, а не символ операции конкатенации строк*.

Большинство методов класса `String` имеет входные параметры, на значения которых наложены ограничения. Корректное выполнение метода возможно только в том случае, когда фактические значения параметров, передаваемые методу при его вызове, удовлетворяет заданным ограничениям.

При помощи метода `charAt` можно получить один из символов текущей строки. Метод имеет один входной параметр типа `int`, при помощи которого ему передается индекс интересующего нас символа. Метод возвращает значение типа `char`, представляющее собой символ строки, индекс которого был передан методу в качестве входного параметра. Для корректного выполнения метода `charAt` значение его входного параметра не может быть отрицательным числом либо положительным числом, величина которого превышает количество символов в строке. На рис. 4.9 приведены два фрагмента кода, иллюстрирующие применение метода `charAt`.

Первое предложение, на рис. 4.9 а), создает строку с именем `word` и значением `"кот"`. Последующие три предложения выделяют из этой строки первый, второй и третий символы и присваивают их значения в переменные `firstLetter`, `secondLetter` и `thirdLetter` соответственно. Выделение символов из строки осуществляется методом `charAt`. Метод вызывается сообщением `word.charAt(i)`, в котором `i` — индекс символа в строке. Код на рис. 4.9 б) использует оба метода `length` и `charAt`. Его особенностью является то, что в сообщениях, вызывающих методы `length` или `charAt`, в качестве ссылки на объект-строку используется литерал `"кот"`.

```
String word = "кот";
char firstLetter = word.charAt(0);   // в firstLetter символ 'к'
char secondLetter = word.charAt(1); // в secondLetter символ 'о'
char thirdLetter = word.charAt(2);  // в thirdLetter символ 'т'
```

а)

```
int index = "кот".length() - 1;     // в index число 2
char letter = "кот".charAt(index);  // в letter символ 'т'
```

б)

**Рис. 4.9.** Примеры использования метода `charAt`

При помощи метода substring можно выделить подстроку из строки. В классе String метод substring представлен двумя вариантами, отличающимися списком входных параметров и кодом.

Первый из методов substring, приведенных на рис. 4.6, имеет один входной параметр типа int, при помощи которого методу передается индекс символа, начиная с которого формируется подстрока. Метод возвращает подстроку, выделяемую из текущей строки, начиная с символа заданного входным параметром и до конца строки.

Второй из методов substring имеет два параметра типа int. Параметр с именем begin задает индекс символа, начиная с которого формируется подстрока. Параметр с именем end задает индекс символа, до которого формируется подстрока. Метод возвращает подстроку, выделяемую из текущей строки, начиная с символа, заданного первым параметром begin, и *до символа*, заданного вторым параметром end (не включая его). На рис. 4.10 приведены предложения, иллюстрирующие применение метода substring.

```
String sentence = "Евгений Онегин";
String word1, word2, word3;
   word1 = sentence.substring(0,7); // в word1 "Евгений"
   word2 = sentence.substring(2,7); // в word2 "гений"
   word3 = sentence.substring(8);   // в word3 "Онегин"
```

**Рис. 4.10.** Примеры выделения подстрок при помощи метода substring

Первое предложение на рис. 4.10 создает строку с именем sentence и значением "Евгений Онегин". Второе предложение объявляет три ссылочные переменные с именами word1, word2, и word3 типа String. Отметим, что при помощи этого предложения не создаются строки (объекты класса String), а лишь ссылочные переменные. Соответствующие строки создаются при выполнении последующих трех предложений, из которых в первых двух предложениях вызывается метод substring с двумя входными параметрами, а в последнем — метод substring с одним входным параметром. Например, для того чтобы создать строку word1 со значением "Евгений" из текущей строки sentence со значением "Евгений Онегин" необходимо методу substring передать значение индекса первого символа текущей строки (символ 'E' с индексом 0) и значение индекса текущей строки, до которого выделяется подстрока (символ ' ' (пробел) с индексом 7).

Комбинируя методы, объявленные в классе String, можно создавать новые методы манипулирования строковыми данными. Рассмотрим, в качестве примера, каким образом можно выделить средний символ из заданной строки символов. Понятие «средний символ» применимо для строк, состоящих из нечетного количества символов. Например, для строки "паровоз" средним символом будет символ 'o'. Ясно, что выделить средний символ из строки можно при помощи метода charAt. Задача заключается в том, что необходимо предложить способ вычисления

значения параметра для метода charAt, задающего индекс среднего символа. Решить эту задачу можно с использованием метода length. На рис. 4.11 приведены варианты кода, при помощи которого можно получить средний символ строки.

```
String word = "паровоз";           // предложение 1
char midSymbol;                    // предложение 2
int midIndex = word.length()/2;    // предложение 3
midSymbol = word.charAt(midIndex); // предложение 4
```

а)

```
String word = "паровоз";
char midSymbol = word.charAt(word.lenghh()/2);
```

б)

**Рис.4.11.** Варианты кода для получения среднего символа строки

Предложение 1 на рис. 4.11а создает строку с именем word и значением "паровоз", средний символ которой мы намерены получить. Предложение 2 объявляет переменную с именем midSymbol типа char, в которую, в итоге, мы должны поместить найденный средний символ строки word. В левой части предложения 3 объявлена целочисленная переменная с именем midIndex и типом int, предназначенная для хранения индекса среднего символа. В правой части предложения 3 записано выражение, осуществляющее вычисление индекса среднего символа. Сообщение word.length() в этом выражении вызывает метод length, который возвращает длину строки word в виде целого числа типа int. Полученное значение длины строки делится на целое число два с отбрасыванием остатка от деления. Читателю предлагается проверить, что для любой строки с нечетным количеством символов и с учетом того, что индекс первого символа строки равен нулю, частное от деления длины строки на два, без учета остатка от деления, всегда равно индексу среднего символа этой строки. Правая часть предложения 4 представляет собой сообщение, вызывающее метод charAt строки word. В качестве параметра методу передается значение переменной midIndex, хранящей, к этому времени, индекс искомого среднего символа. Выделенный методом charAt средний символ присваивается в переменную midSymbol.

Вариант кода, приведенный на рис. 4.11б, отличается от кода на рис. 4.11а тем, что объявление переменной midSymbol, а также действия по вычислению индекса среднего символа и выделению среднего символа из строки word закодированы в одном предложении. Особенностью этого предложения является то, что параметром метода charAt является не переменная или литерал, а значение, возвращаемое выражением word.length()/2.

### 4.3.2. Поиск символов и подстрок в строке

При манипулировании строковыми данными могут возникнуть следующие задачи. Для заданной строки найти месторасположение (индекс) конкретного символа.

Либо для заданной строки найти индекс, начиная с которого в строке располагается конкретная подстрока. В состав класса `String` входит несколько методов, позволяющих решить отмеченные задачи. Одним из наиболее употребительных является метод `indexOf`, существующий в четырех вариантах, отличающихся входными параметрами и кодом. На рис. 4.12 приведены заголовки отмеченных вариантов метода `indexOf`.

Первые два метода `indexOf` на рис. 4.12 предназначены для поиска индекса символа, заданного значением параметра `symb`, а третий и четвертый — для поиска индекса подстроки, заданной значением параметра `subStr`. Особенностью объявления методов `indexOf` в классе `String` является то, что типом параметра `symb` является целочисленный тип `int`, а не символьный тип `char`. Однако, как это было отмечено ранее, тип `int` позволяет представлять Unicode-номер любого знака базовой многоязыковой плоскости пространства Unicode. При вызове метода `indexOf` фактическими значениями параметра `symb` являются, обычно, символьные переменные или литералы.

```
public int indexOf(int symb)
public int indexOf(int symb, int begin)
public int indexOf(String subStr)
public int indexOf(String subStr, int begin)
```

**Рис. 4.12.** Заголовки метода `indexOf`

Для корректного использования методов `indexOf` необходимо следить, чтобы индексы, передаваемые в качестве входных параметров, были положительными числами, не превышающими индекс последнего символа строки. В том случае, когда метод `indexOf` не обнаруживает в строке требуемый символ или подстроку, он возвращает значение –1.

Первый вариант метода `indexOf` на рис. 4.12 имеет один входной параметр с именем `symb`, задающий значение символа, индекс которого разыскивается методом. Метод возвращает индекс первого вхождения `symb` в текущую строку.

Второй вариант метода `indexOf` на рис. 4.12 имеет два входных параметра. Параметр `symb` задает значение символа, индекс которого разыскивается методом. Параметр `begin` задает индекс символа строки, начиная с которого осуществляется поиск. Метод возвращает индекс первого вхождения `symb` в текущую строку, который расположен правее символа с индексом `begin`.

Третий вариант метода `indexOf` на рис. 4.12 имеет один входной параметр с именем `subStr`, задающий значение подстроки, индекс которой разыскивается методом. Метод возвращает индекс первого символа подстроки `subStr` в текущей строке.

Четвертый вариант метода `indexOf` на рис. 4.12 имеет два входных параметра. Параметр `subStr` задает значение подстроки, индекс которой разыскивается методом. Параметр `begin` задает индекс символа текущей строки, начиная с

которого осуществляется поиск. Метод возвращает индекс первого символа подстроки subStr, расположенной правее символа текущей строки с индексом begin.

Предложения на рис. 4.13 иллюстрируют применение методов indexOf.

```
int location;
location = "Программирование".indexOf('o');      // предл. 1, в location 2
location = "Программирование".indexOf('o', 4);   // предл. 2, в location 10
location = "Программирование".indexOf("po");     // предл. 3, в location 1
location = "Программирование".indexOf("po", 4);  // предл. 4, в location 9
location = "Программирование".indexOf("po", 11); // предл. 5, в location -1
```

**Рис. 4.13.** Примеры использования методов indexOf

Первое предложение на рис. 4.13 объявляет целочисленную переменную с именем location, которая затем используется в левой части предложений 1–5. Правая часть предложений 1–5 представляет собой сообщение, адресованное строке "Программирование" и вызывающее метод indexOf.

В предложении 1 параметром метода indexOf является символьный литерал 'o'. Метод разыскивает в строке "Программирование" символ 'o', сканируя ее слева направо, начиная с первого символа, и возвращает число 2, которое является индексом первого символа 'o' в строке "Программирование".

В предложении 2 вызывается метод indexOf с двумя параметрами. Первый параметр опять задает символьный литерал 'o', а второй — индекс символа строки, начиная с которого осуществляется сканирование. Метод разыскивает в строке "Программирование" символ 'o', сканируя строку слева направо, начиная с символа, имеющего индекс 4 (символ 'p'), и возвращает число 10, которое является индексом символа 'o', обнаруженного правее символа с индексом 4.

В предложении 3 параметром метода indexOf является строковый литерал "po". Метод разыскивает в строке "Программирование" подстроку "po", сканируя ее слева направо, начиная с первого символа, и возвращает число 1, которое является начальным индексом первого появления подстроки "po" в строке "Программирование".

В предложении 4 вызывается метод indexOf с двумя параметрами. Первый параметр задает разыскиваемую подстроку в виде литерала "po", а второй — индекс символа строки, начиная с которого осуществляется сканирование. Метод сканирует строку слева направо, начиная с символа, имеющего индекс 4, и возвращает число 9, которое является начальным индексом подстроки "po", обнаруженной правее символа, имеющего индекс 4.

Предложение 5 иллюстрирует случай, когда значения входных параметров метода indexOf специфицируют отсутствующую подстроку. Как было сказано ранее, если метод не обнаруживает в строке требуемый символ или подстроку, он возвращает значение –1.

### 4.3.3. Сравнение содержимого строк

Поскольку строки состоят из символов, а символы могут сравниваться между собой, то естественно распространить возможность сравнения отдельных символов и на строки символов. Класс `String` содержит некоторое количество методов, предназначенных для сравнения строк.

Две строки могут сравниваться между собой с целью определения их порядка следования в случае сортировки списка строк. Сортироваться может, например, список фамилий студентов с целью расположения их в алфавитном порядке. При этом, как и в случае сравнения символов, словосочетание «строка Y больше строки X» следует понимать как «строка Y располагается после строки X». Основой для сравнения строк является сравнение соответствующих символов этих строки. При сравнении двух строк вначале сравниваются первые символы этих строк. На первое место ставится строка, у которой первый символ имеет меньшее значение. Если первые символы одинаковы, то сравниваются вторые символы обеих строк и т. д.

Особенностью работы методов сравнения строк, объявленных в классе `String`, является использование при сравнении символов только их Unicode-номеров. В некоторых случаях это приводит к тому, что расположение строк в отсортированном списке может не совпадать с их расположением в соответствии с грамматикой национального языка. Это, как правило, происходит при сортировке строк для языков, изобилующих диакритическими знаками (знаки, располагающиеся над и под символами).

На рис. 4.14 приведены заголовки некоторых методов класса `String`, предназначенных для сравнения строк.

```
public boolean equals(Object str)
public boolean equalsIgnoreCase(String str)
public int compareTo(String str)
public int compareToIgnoreCase(String str)
```

**Рис. 4.14.** Заголовки методов сравнения строк

При помощи метода `equals` можно узнать, являются ли две строки эквивалентными или нет. Особенностью объявления метода `equals` в классе `String` является то, что входной параметр метода `equals` имеет тип `Object`, а не тип `String`. Класс `Object` — это класс-предок для класса `String`. При практическом использовании метода `equals` в качестве входного параметра используется ссылка на объект класса `String`. Отношение между классами-предками и классами-потомками будет подробно изучаться в разделах, посвященных наследованию. Метод `equals` последовательно сравнивает Unicode-номера символов текущей строки с Unicode-номерами символов строки, ссылка на которую передается ему в качестве входного параметра с именем `str`. Метод возвращает значение `true`, если обе строки имеют одинаковую длину и состоят из символов с тождественными Unicode-номерами. При сравнении

учитывается тот факт, что Unicode-номера строчных (малых) и прописных (больших) букв имеют различное значение.

Метод `equalsIgnoreCase` работает точно так же, как и метод `equals`, однако игнорирует различие в Unicode-номерах строчного и прописного варианта одной и той же буквы, считая их равными. Входной параметр метода `equalsIgnoreCase` имеет тип `String`.

Метод `compareTo` сравнивает содержимое текущей строки со строкой, ссылка на которую передается методу в качестве входного параметра `str`. Метод возвращает целочисленное значение типа `int`. Возвращаемое значение является отрицательным числом, если строка, переданная методу при помощи входного параметра, предшествует текущей строке. Возвращаемое значение является положительным числом, если строка, переданная методу при помощи входного параметра, следует за текущей строкой. Возвращаемое значение равно нулю, если строки одинаковы (случай, когда метод `equals` возвращает значение `true`). Строки сравниваются лексикографически. Это означает, что две строки, имеющие одинаковое количество символов, считаются различными, если символы с одним и тем же индексом в обеих строках имеют различное значение Unicode-номера. Строки считаются различными и в том случае, если они состоят из разного количества символов. В этом случае строка, имеющая меньшее количество символов, предшествует строке, имеющей большее количество символов.

Метод `compareToIgnoreCase` работает точно так же, как и метод `compareTo`, однако игнорирует различие в Unicode-номерах строчного и прописного варианта одной и той же буквы, считая такие буквы одинаковыми. На рис. 4.15 приведен пример кода, который иллюстрирует применение методов сравнения содержимого строк.

```
String firstGirl = "Арина";
String secondGirl = "Ирина";
boolean who;
int which;
   who = "Арина".equals(firstGirl);         // в who значение true
   who = "Арина".equals(secondGirl);        // в who значение false
   which = "Ирина".compareTo(firstGirl);    // в which положительное число
   which = "Арина".compareTo(secondGirl); // в which отрицательное число
   which = "Арина".compareTo(firstGirl);    // в which число 0
```

**Рис. 4.15.** Примеры использования методов сравнения содержимого строк

Первые два предложения на рис. 4.15 создают строки с именами `firstGirl`, `secondGirl` и значениями `"Арина"`, `"Ирина"`, соответственно. Значения строк отличаются первым символом, из которых символ `'А'` имеет меньший Unicode-номер, чем символ `'И'`. Третье и четвертое предложения на рис. 4.15 объявляют переменные с именами `who`, `which` и типами `boolean`, `int`, соответственно. В дальнейшем мы

использует переменную who для хранения возвращаемого значения метода equals, а переменную which для хранения возвращаемого значения метода compareTo. Пятое и шестое предложения на рис. 4.15 иллюстрируют работу метода equals, вызываемого из стоки "Арина". В том случае, когда методу equals передается имя строки firstGirl, он сравнивает тождественные строки и возвращает значение true. В том случае, когда методу equals передается имя строки secondGirl, он сравнивает различные строки и возвращает значение false. Последние три предложения на рис. 4.15 иллюстрируют работу метода compareTo. Когда метод compareTo вызывается из строки "Ирина", а входным параметром является строка с именем firstGirl и значением "Арина" (седьмое предложение на рис. 4.12), метод возвращает положительное число, которое следует интерпретировать как: «строка, указанная в параметре метода, предшествует текущей строке». В восьмом предложении на рис. 4.15 метод compareTo сравнивает строку с именем secondGirl и значением "Ирина" с текущей строкой "Арина". Метод возвращает отрицательное число, которое интерпретируется как: «строка, указанная в параметре метода, следует за текущей строкой». Последнее, девятое, предложение на рис. 4.15 иллюстрирует случай, когда метод compareTo сравнивает две одинаковые строки. В этом случае метод возвращает число ноль.

### 4.3.4. Методы обработки строк

Класс String включает некоторое количество методов, предназначенных для манипулирования текущей строкой при помощи таких операций, как удаление лишних пробелов, расположенных перед первым символом строки и после последнего символа строки; замена всех вхождений некоторого символа на новый символ и т. п. Здесь уместно напомнить, что характерной особенностью всех методов, объявленных в классе String, является то, что они не изменяют значение строки, к которой применяются, и, следовательно, методы обработки строк, рассматриваемые в настоящем подразделе, возвращают новую строку, оставляя исходную строку «неповрежденной».

На рис. 4.16 приведены заголовки некоторых методов обработки строк, объявленных в классе String.

```
public String trim()
public String concat(String str)
public String replace(char oldChar, char newChar)
```

**Рис. 4.16.** Заголовки методов обработки строк

Метод trim создает строку, отличающуюся от текущей строки тем, что в ней удалены все пробелы, если таковые имеются, которые расположены перед первым символом строки и после последнего символа строки. Пробелы,

расположенные внутри строки, не удаляются. Код на рис. 4.17 иллюстрирует применение метода `trim`.

```
String withSpaces = "  два объекта  ";
String withoutSpaces = withSpaces.trim(); // в withoutSpaces "два объекта"
```

**Рис. 4.17.** Пример использования метода `trim`

Первое предложение на рис. 4.17 создает строку с именем `withSpaces` и значением, содержащим по два пробела перед первым символом и после последнего символа. Во втором предложении на рис. 4.17 создается новая строка с именем `withoutSpaces`, в которую помещается значение текущей строки `withSpaces` без пробелов перед первым символом и после последнего символа, удаленных методом `trim`.

Метод `concat` осуществляет конкатенацию двух строк: текущей строки и строки, ссылка на которую передается методу при помощи входного параметра `str` (см. рис. 4.16). Метод `concat` действует точно так же, как и операция конкатенации строк в виде символа «+». На рис. 4.18 приведен код, иллюстрирующий работу метода `concat`.

```
String first = "Это ";                                          // предложение 1
String second = "я знаю";                                       // предложение 2
String third, fourth;                                           // предложение 3
  third = first.concat(second); // в third "Это я знаю"         // предложение 4
  fourth = third.concat(" и помню").concat(" прекрасно");       // предложение 5
  fourth = first + second + " и помню" + " прекрасно";          // предложение 6
  // в fourth строка "Это я знаю и помню прекрасно"
```

**Рис. 4.18.** Пример использования метода `concat`

Предложения 1 и 2 на рис. 4.18 создают строки с именами `first`, `second` и значениями `"Это "`, `"я знаю"`, соответственно. Предложение 3 объявляет две ссылочные переменные `third`, `fourth` типа `String`. Предложение 4 создает строку, ссылка на которую помещается в переменную `third`. Строка создается путем конкатенации строки с именем `first` со строкой с именем `second`. Операция конкатенации выполняется методом `concat`, который вызывается сообщением `first.concat(second)`. После выполнения предложения 4 значение строки `third` становится равным `"Это я знаю"`. Действия, закодированные в правой части пятого предложения, иногда называют *каскадной конкатенацией*. При каскадной конкатенации метод `concat` применяется несколько раз последовательно. Текущей строкой становится строка, полученная в результате выполнения предыдущей конкатенации. Эта строка подвергается последующей конкатенации очередным методом `concat`. Первый вызов метода `concat`, в правой части предложения 5,

формирует строку со значением "Это я знаю и помню". Ссылка на эту строку используется для вызова второго метода concat. В результате работы этого метода создается окончательная строка с именем fourth и значением "Это я знаю и помню прекрасно". Предложение 6 показывает, как можно получить то же самое значение окончательной строки с использованием символа операции конкатенации «+», рассмотренного ранее.

Метод replace создает новую строку, отличающуюся от текущей тем, что в ней изменены все вхождения некоторого символа на новый символ. Символ, который необходимо заменить, передается методу при помощи входного параметра с именем oldChar, а новый символ — при помощи входного параметра с именем newChar (см. рис. 4.16). На рис. 4.19 приведен код, иллюстрирующий применение метода replace.

```
String initial = "Пишу без ашибак";
String result;
   result = initial.replace('a', 'o') // в result строка "Пишу без ошибок"
```

**Рис. 4.19.** Пример использования метода replace

## Упражнения для самостоятельной работы

4.1. Разработайте программный код для вычисления средней длины трех произвольных строк.

4.2. Разработайте программный код для выполнения следующих действий над двумя произвольными строками: (1) присвойте булевой переменной результат проверки на равенство данных строк; (2) присвойте булевой переменной результат проверки на равенство данных строк, не учитывая различие между прописными и строчными буквами.

4.3. Разработайте программный код для выполнения следующих действий над двумя произвольными строками: (1) присвойте целочисленной переменной результат проверки на равенство данных строк; (2) присвойте целой переменной результат проверки на равенство данных строк, не учитывая различие между прописными и строчными буквами; (3) создайте новую строку путем сцепления строк.

4.4. Разработайте программный код для выполнения следующих действий над строкой: (1) присвойте символьной переменной первый символ строки; (2) присвойте целой переменной индекс первого заданного символа в строке.

4.5. Разработайте программный код для выполнения следующих действий над строкой `s1`, имеющей значение `"This is starting literal"`: (1) выделите из строки подстроку, начиная с индекса 1; (2) выделите из строки подстроку, начиная с индекса 8 и до индекса 14.

4.6. Разработайте программный код для выполнения следующих действий над строкой `s1`, имеющей значение `"  This is starting literal  "`: (1) создайте новую строку путем удаления пробелов с обоих концов строки `s1`; (2) замените в строке `s1` все вхождения символа `'s'` на символ `'S'`. Результат запишите в новую строку.

4.7. Разработайте программный код, который из строк `"Welcome Java"` и `"to"` сформирует новую строку `"Welcome to Java"`.

4.8. Разработайте программный код, который из строки `"Казнить, нельзя помиловать"` сформирует новую строку `"Канить нельзя, помиловать"`.

4.9. Разработайте программный код, который из строки `"Я помню чудное мгновение… А.С.Пушкин"` составит новую строку: `"Я помню чудное мгновение… (А.С.Пушкин)"`.

4.10. Разработайте программный код, который из строки, содержащей имя, отчество и фамилию студента, выделит только его фамилию.

4.11. Используя код, полученный при выполнении упражнения 4.9, разработайте программу, которая из строки, содержащей имя, отчество и фамилию студента, выделит инициалы и сформирует новую строку, состоящую из фамилии и инициалов. После каждого инициала необходимо расположить точку.

4.12. Разработайте программный код, который по полному имени файла определит, является ли данный файл исходным кодом на языке программирования Java.

Раздел 5

# КОНСОЛЬНЫЙ ВВОД И ВЫВОД ДАННЫХ

Вводом данных будем называть *чтение данных из внешнего источника данных*, а выводом — *запись данных во внешний получатель данный*. Источник и получатель данных являются внешними по отношению к центральной части компьютера. В случае ввода, данные перемещаются из внешнего источника в переменные компьютерной программы, а в случае вывода — из переменных компьютерной программы во внешний получатель данных.

Часто источником и получателем данных при вводе и выводе являются файлы данных, расположенные на внешних носителях. Однако, в общем случае, ввод и вывод не ограничиваются только файлами. Источником и получателем данных могут быть также объекты классов, например, строки данных (объекты класса `String`) или массивы данных (объекты классов массивов) и другие. Возможна комбинация различных видов источников и получателей данных. Например, данные могут читаться из файла и записываться в объект класса `String`.

Алгоритмы программ часто предполагают как ввод, так и вывод данных. Однако это не обязательно. Программа может, например, генерировать большое количество данных и записывать их в файл. В этом случае используется только вывод данных, а ввод данных отсутствует.

Внешние источники и получатели данных «материализуются» различными периферийными устройствами компьютера. Среди множества периферийных устройств выделяют два устройства, используемые операционной системой для ввода и вывода данных. Эти устройства называются *системным устройством ввода* и *системным устройством вывода*. В современных компьютерах общего назначения в качестве системного устройства ввода используется клавиатура, а в качестве системного устройства вывода — экран монитора. *Системные устройства ввода и вывода часто называют консолью*. Таким образом, консоль включает *клавиатуру консоли* и *экран консоли*. Экран консоли используется не только как внешний получатель данных от прикладной программы, но и как устройство, на которое операционная система выводит свои сообщения, в частности, сообщения об ошибках.

## 5.1. Потоки данных при вводе и выводе

В основе логической организации системы ввода и вывода данных лежит *концепция потоков данных*. Концепция потоков данных представляет перемещение данных между компьютерной программой и внешним источником/получателем данных в виде непрерывного потока дискретных порций данных. Порциями данных в потоке могут быть байты, символы, объекты либо иные логические элементы данных. Рис. 5.1 иллюстрирует концепцию потоков данных.

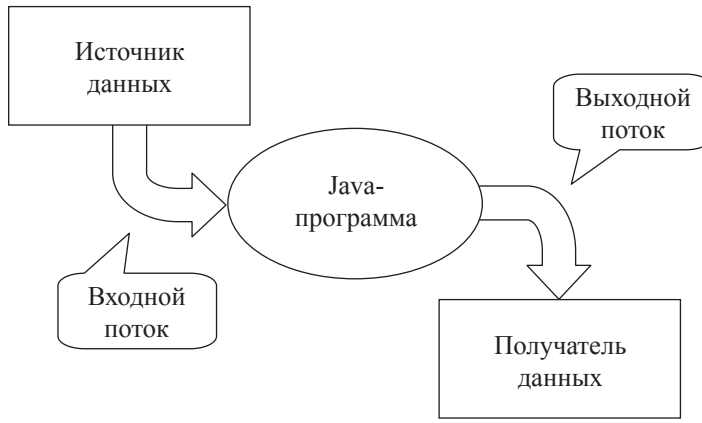



**Рис. 5.1.** Входной и выходной потоки данных

Как это видно на рис. 5.1, в системе ввода и вывода данных существуют два потока данных. Поток, который связывает источник данных с программой, называется *входным потоком*, а поток, который связывает программу с получателем данных, — *выходным потоком*. Используя понятия входной и выходной потоки данных, ввод можно рассматривать как *чтение данных из входного потока*, а вывод данных — как *запись данных в выходной поток*.

Для организации ввода данных из источника данных необходимо, в общем случае, выполнить следующие действия.

- Идентифицировать источник данных: файл, строка, массив, логический сетевой коннектор либо иной источник данных.
- Создать входной поток, используя идентифицированный источник данных, и открыть его для чтения. В языке программирования Java создание входного потока автоматически открывает его для чтения.
- Читать данные из входного потока. Обычно данные из входного потока читаются в итерационном процессе, в котором на каждой итерации читается одна порция данных. Методы, осуществляющие чтение из входного потока, возвращают значение, индицирующее отсутствие данных во входном потоке, которое можно использовать для определения момента завершения итераций.
- Закрыть входной поток.

Для вывода данных получателю необходимо, в общем случае, выполнить следующие действия.

- Идентифицировать получателя данных. Как и в случае ввода, это может быть файл, строка, массив, логический сетевой коннектор либо иной получатель.
- Создать выходной поток, используя ранее идентифицированного получателя данных. Создание выходного потока автоматически открывает его для записи.
- Записывать данные в выходной поток при помощи итерационного процесса, в котором на каждой итерации в выходной поток записывается одна порция данных.
- Закрыть выходной поток.

Для реализации перечисленных действий в программном коде обычно используются методы предопределенных классов, моделирующих входные и выходные потоки данных. Поэтому создание входного или выходного потока означает создание объекта соответствующего класса. Наличие большого количества предопределенных классов, моделирующих входные и выходные потоки, не исключает, в случае необходимости, создание своих собственных классов. Таким образом, язык программирования Java допускает создание и использование новых классов, моделирующих новые типы входных и выходных потоков данных. Такая возможность может понадобиться в том случае, когда источник и получатель данных реализуются нестандартным периферийным оборудованием компьютера.

## 5.2. Вывод данных на консоль

При изучении основ императивного программирования и навыков кодирования методов, как правило, в качестве внешнего источника и получателя данных рассматривают системные устройства ввода и вывода.

Для того, чтобы организовать вывод данных из программы в выходной поток, связанный с консолью, не обязательно «вручную» кодировать действия, перечисленные в подразделе 5.1. В языке программирования Java существует предопределенный класс с именем `PrintStream`, содержащий методы, реализующие эти действия. В классе `PrintStream` объявлены методы, специализированные на создание выходного потока, состоящего из последовательности символов. Методы класса `PrintStream` реализуют только основные функции консоли, достаточные для отображения данных из входного потока на экран консоли. Функции консоли, которые не являются существенными для отображения данных из входного потока на экране консоли, не реализуются методами класса `PrintStream`. Так, например, в классе `PrintStream` отсутствует метод, позволяющий изменить цвет фона экрана.

Для того, чтобы воспользоваться методами класса `PrintStream` для кодирования вывода данных на консоль, необходимо знать:

1. имя объекта класса `PrintStream`, моделирующего экран консоль;
2. список, назначение и особенности работы каждого из предопределенных методов;
3. структуру сообщения, вызывающего требуемый метод.

### 5.2.1. Объект `System.out`

Кроме предопределенного класса `PrintStream`, в системе программирования Java существует также предопределенный объект этого класса с именем `System.out`. Объект с именем `System.out` моделирует экран консоли, и методы этого объекта могут выполнять действия по выводу данных в выходной поток. Объект `System.out` *существует в системе и его не надо создавать* перед вызовом методов класса `PrintStream`.

Имя объекта `System.out` является, по сути, именем ссылочной переменной, которая содержит адрес участка памяти, в котором расположен сам объект, моделирующий экран консоли. Рис. 5.2 иллюстрирует отношение между ссылочной переменной `System.out`, объектом класса `PrintStream`, моделирующим экран консоли, и физическим экраном консоли.

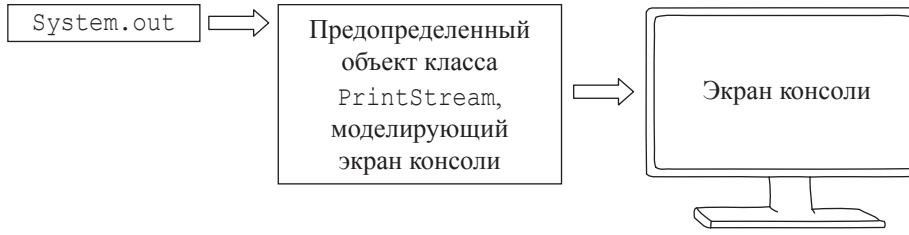



**Рис. 5.2.** Моделирование монитора консоли предопределенным объектом класса `PrintStream`

Имя ссылочной переменной `System.out` является составным и составлено из имени предопределенного класса `System` и имени его статического поля `out`. В классе `System` поле `out` объявлено при помощи следующего предложения

```
public static final PrintStream out;
```

Классификация полей классов и специфика статических полей будет изучаться во второй части книги, в разделе, посвященном моделированию классов. Пока что отметим, что имя статического поля всегда является составным и включает имя класса, в котором объявлено статическое поле.

Инициализацию поля `out` осуществляет виртуальная машина Java. Виртуальная машина Java инициализирует поле `out` значением, соответствующим адресу предопределенного объекта, моделирующего экран консоли конкретного компьютера.

### 5.2.2. Методы класса `PrintStream`

В классе `PrintStream` объявлены методы, позволяющие осуществлять вывод данных из Java-программы на экран консоли через выходной поток. Для удобства изучения методов класса `PrintStream` их целесообразно объединить в группы. В практике императивного программирования чаще всего используются методы из следующих трех групп:

- методы с именем `print`;
- методы с именем `println`;
- методы с именем `printf`.

Методы в каждой из групп имеют одинаковое имя, но различные типы параметров и называются *перегруженными*. Группа методов с именем `print` состоит из 9 методов со следующими заголовками:

```
public void print(boolean b)
public void print(char c)
public void print(int i)
public void print(long l)
public void print(float f)
public void print(double d)
public void print(char[] s)
public void print(String s)
public void print(Object obj)
```

Набор перегруженных методов `print` позволяет выводить на экран консоли данные встроенных типов, а также массивов, строк и объектов классов, определяемых программистом. Последняя возможность реализована последним методом в приведенном списке. `Object` является именем предопределенного класса, возглавляющего иерархию наследования всех классов в языке программирования Java. Это означает, что члены класса `Object` наследуются всеми остальными классами, и поэтому тип `Object` может вербально интерпретироваться как «любой объект». Идея наследования и вопросы проектирования иерархии наследования изучаются во второй части книги.

Отличительной характеристикой методов `print` является то, что все данные, специфицированные параметрами метода, выводятся в одной строке, а после завершения работы метода курсор не перемещается к началу следующей строки. Поэтому, если, например, последовательно вызываются два метода `print`, то

второй метод будет продолжать вывод данных в ту же строку, в которую осуществлял вывод первый метод.

Группа методов с именем `println` состоит из 10 методов:

```
public void println(boolean b)
public void println(char c)
public void println(int i)
public void println(long l)
public void println(float f)
public void println(double d)
public void println(char[] s)
public void println(String s)
public void println(Object obj)
public void println()
```

Первые девять методов в приведенном списке работают почти так же, как методы `print`. Отличие заключается в том, что после завершения вывода курсор перемещается к началу следующей строки. Поэтому, если, например, последовательно вызываются два метода `println`, то второй метод будет выводить данные с начала следующей строки. Десятый метод не имеет входных параметров, и его работа заключается в установке курсора в начало следующей строки.

Группа методов с именем `printf` состоит из 2 методов со следующими заголовками:

```
public PrintStream printf(String format, Object... args)
public PrintStream printf(Local loc, String format, Object... args)
```

Методы `printf` позволяют осуществлять форматированный вывод данных. Форматированный вывод означает, что программист управляет расположением данных в поле вывода строки путем описания этого расположения при помощи *спецификаторов формата*. Каждому выводимому данному ставится в соответствие индивидуальный спецификатор формата, задаваемый параметром `format`. Строка, содержащая спецификаторы формата, задаваемая параметром `format`, должна быть согласована со списком выводимых данных, задаваемых параметром `args`.

Язык программирования Java является не только многоплатформенным, но и многоязычным. Средства обеспечения многоязычности частично обсуждались при изучении системы Unicode (см. подраздел 3.3.2). Входной параметр метода `printf` с именем `loc` также является одним из средств, обеспечивающих многоязычность, и позволяет выполнить форматированный вывод с учетом языковой специфики географического региона. Если этот параметр отсутствует, то языковая локализация не производится.

### 5.2.3. Вызов методов класса PrintStream

Вызов методов класса PrintStream осуществляется при помощи сообщений. Со структурой сообщений мы уже познакомились при изучении методов класса String. Как было отмечено, любое сообщение имеет следующую структуру.

```
<ссылка на объект> . <имя метода>(<список фактических параметров>)
```

Поскольку группы методов print, println и printf класса PrintStream перегружены и, следовательно, имеют одинаковое имя, то *вызов любого из методов в пределах группы осуществляется при помощи одного сообщения*. Поэтому для кодирования вывода данных на экран консоли при помощи методов print, println и printf достаточно четырех сообщений.

```
System.out.print(<спецификации выводимых данных>)          (1)
System.out.println(<спецификации выводимых данных>)        (2)
System.out.println()                                       (3)
System.out.printf(<спецификации выводимых данных>)         (4)
```

Первое и второе сообщения вызывают один из перегруженных методов print или println. Выбор конкретного метода из группы перегруженных методов детерминируется типом выводимых данных. Третье сообщение вызывает метод, переводящий курсор к началу следующей строки. Четвертое сообщение вызывает один из перегруженных методов printf.

На рис. 5.3 приведен фрагмент кода, иллюстрирующий использование методов print и println для вывода данных на экран консоли, а также строки, выводимые на экран в результате работы кода.

```
int numb = 2;
double root = Math.sqrt(numb);
System.out.print("Корень квадратный числа ");
System.out.print(numb);
System.out.print(" равен ");
System.out.print(root);
System.out.println(".");
numb = 5;
root = Math.sqrt(numb);
System.out.println("Корень квадратный числа " + numb +
                   " равен " + root + ".");
Корень квадратный числа 2 равен 1.4142135623730951.
Корень квадратный числа 5 равен 2.2360679774997914.
```

**Рис. 5.3.** Примеры использования методов print и println

Фрагмент кода на рис. 5.3 состоит из двух частей. В первой части кода вначале объявляется целочисленная переменная с именем numb (число), которая инициализируется числом 2. Затем объявляется переменная с именем root (корень) типа double. Эта переменная инициализируется квадратным корнем числа 2. Квадратный корень вычисляется методом с именем sqrt, который вызывается из предопределенного класса Math. Метод sqrt является статическим, поэтому он вызывается сообщением, состоящим из имени класса и имени метода. Статические методы будут изучаться во второй части книги, в разделе, посвященном моделированию классов. Затем следует пять предложений, состоящих из сообщений, вызывающих методы print и println. Выполнение этих предложений формирует на экране консоли следующую строку.

```
Корень квадратный числа 2 равен 1.41421356237309.
```

Предложения вызывают один из перегруженных методов print или println класса String в соответствии с типом фактических параметров. Например, сообщение System.out.print("Корень квадратный числа ") вызывает метод, который выводит на консоль объект класса String, а сообщение System.out.print(numb) вызывает метод, который выводит на экран консоли значение переменной типа int. Пятое сообщение вызывает метод println, который после вывода символа «точка» переводит курсор к началу следующей строки.

Методы print и println допускают в качестве фактических параметров использовать разнотипные данные, связанные операцией конкатенации. Поскольку операция конкатенации применима для строковых данных (объектов класса String), то перед выполнением конкатенации все данные автоматически преобразуются в строковые данные. Это преобразование осуществляется транслятором на этапе формирования байт-кода. Поэтому вместо последовательного вызова нескольких методов print и println можно вызвать только один из них, сформировав выводимую строку в виде последовательности данных, связанных операцией конкатенации. Эта возможность иллюстрируется второй частью фрагмента кода, приведенного на рис. 5.3. Во второй части кода вначале значение переменной numb изменяется на число 5, а переменной root присваивается значение квадратного корня числа 5. Затем на экран консоли выводится строка.

```
Корень квадратный числа 5 равен 2.23606797749979.
```

Вывод этой строки кодируется при помощи одного предложения.

```
System.out.print("Корень квадратный числа " + numb +
                 " равен " + root + ".");
```

В списке фактических параметров методов print и println операндами операции конкатенации могут быть не только разнотипные данные, но и выражения, а также сообщения, осуществляющие вызовы методов. При использовании

арифметических выражений в списке фактических параметров методов `print` и `println` следует следить за тем, чтобы код оставался понятным и легко читаемым, а также за тем, чтобы операция конкатенации явно отличалась от арифметической операции сложения. Обе эти операции кодируются символом «+», и для их различения целесообразно арифметические выражения заключать в круглые скобки. Если скобки отсутствуют, то действует правило умолчания, согласно которому конкатенация имеет более высокий приоритет, чем операция сложения. Поэтому при отсутствии скобок транслятор воспринимает символ «+» как операцию конкатенации. На рис. 5.4 приведен фрагмент кода, иллюстрирующий использование арифметических выражений в списке параметров `print` и `println`.

```
int x = 10,
    y = 200;
System.out.println("x + y = " + x + y);
System.out.println("x + y = " + (x + y));

x + y = 10200
x + y = 210
```

**Рис. 5.4.** Использование арифметических выражений в списке фактических параметров методов `print` и `println`

Первое предложение кода на рис. 5.4 объявляет и инициализирует две целочисленные переменные с именами `x` и `y`. Затем дважды вызывается метод `println`. При первом вызове метода `println` ему передается список фактических параметров в виде

```
("x + y = " + x + y)
```

В приведенном списке параметров транслятор воспринимает два последних символа «+» как символы операции конкатенации, преобразует значения переменных `x` и `y` в строки символов и осуществляет их сцепление. В результате на экран выводится строка `x + y = 10200`. При втором вызове метода `println` ему передается список фактических параметров в виде

```
("x + y = " + (x + y))
```

В приведенном списке параметров транслятор воспринимает выражение в круглых скобках `(x + y)` как арифметическое выражение. При выполнении этого предложения вначале находится сумма значений переменных `x` и `y`, и только после этого полученная сумма преобразуется в символы. В результате на экран выводится строка `x + y = 210`.

Во фрагменте кода на рис. 5.3 методы `print` и `println` использовались для вывода на консоль значений переменной с именем `root` типа `double`. Как видно на рис.

5.3 значение переменной root вывелось со всеми значащими цифрами после запятой. Это является особенностью и недостатком работы методов print и println при выводе значений переменных типов float и double. Такой вывод называется *неформатированным*. При помощи метода printf можно кодировать *форматированный* вывод, который позволяет форматировать строку: выводить числа с плавающей запятой с заданным количества значащих цифр после запятой; выводить числа с плавающей запятой в форме «e»; выравнивать числа по вертикали, если выводится несколько строк чисел и т. д.

### 5.2.4. Форматированный вывод при помощи метода printf

Начнем изучение приемов вывода при помощи метода printf с простого примера, иллюстрирующего различие между неформатированным и форматированным выводом. Пример приведен на рис. 5.5.

```
System.out.println("Число пи = " + Math.PI);
System.out.printf("Число пи = %.2f" ,Math.PI);

Число пи = 3.141592653589793
Число пи = 3.14
```

**Рис. 5.5.** Пример неформатированного и форматированного вывода числа с плавающей запятой

Первое предложение кода на рис. 5.5 вызывает метод println, осуществляющий неформатированный вывод. Список фактических параметров этого метода состоит из строкового литерала "Число пи = " и статического поля с именем PI класса Math, объединенных операцией конкатенации. Поле PI имеет тип double и хранит значение числа π (отношение длины окружности к ее диаметру) с точностью, определяемой типом double. При выполнении первого предложения на экран будут выведены все значащие цифры числа, хранящегося в поле PI. При использовании метода println программист не может управлять количеством выводимых значащих цифр.

Второе предложение кода на рис. 5.5 вызывает метод printf, осуществляющий форматированный вывод. Список фактических параметров метода printf состоит из двух частей. В начальной части в двойных кавычках записывается *форматная строка*, включающая *спецификаторы формата* выводимых данных, а во второй части — список выводимых данных.

На рис 5.5 в форматную строку включен только один спецификатор формата, записанный в виде «%.2f». Этот спецификатор означает, что необходимо вывести соответствующий элемент списка данных (значение поля PI класса Math) в виде числа с плавающей запятой в десятичной системе счисления и с двумя значащими цифрами в дробной части.

В общем случае структура предложения, кодирующего форматированный вывод строк на консоль, имеет вид, приведенный на рис. 5.6.

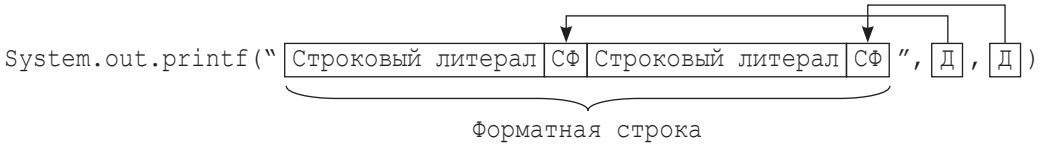



**Рис. 5.6.** Структура предложения, кодирующего форматированный вывод на консоль. СФ — спецификатор формата. Д — данное

В начальной части списка фактических параметров метода `printf` размещается форматная строка, являющаяся, по сути, шаблоном строки, выводимой на экран консоли. Как видно на рис. 5.6, форматная строка представляет собой композицию строковых литералов и спецификаторов формата (СФ). Форматная строка дополняется списком выводимых данных (Д). Разделителем между форматной строкой и списком данных, а также между данными в списке является символ «запятая». Между спецификаторами формата и элементами списка выводимых данных должно быть установлено взаимно однозначное соответствие. На рис. 5.6 это соответствие изображено стрелками. Первый спецификатор формата специфицирует поле вывода первого элемента списка данных, второй спецификатор формата — поле вывода второго элемента списка данных и т. д. При выводе данных на консоль то место форматной строки, где записан СФ, заменяется полем вывода, в которое помещается соответствующее данное в виде, определяемом СФ. На рис. 5.6 структура форматной строки приведена в наиболее общем случае. В частном случае форматная строка может состоять только из спецификаторов формата или только из строкового литерала. В последнем случае список выводимых данных отсутствует, и метод `printf` работает как метод `print` при выводе строкового литерала. Спецификатор формата имеет структуру, приведенную на рис. 5.7.

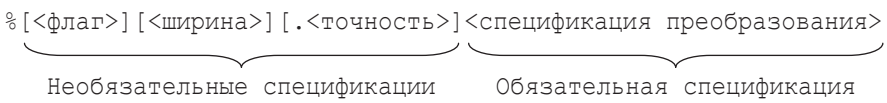



**Рис. 5.7.** Структура спецификатора формата

Спецификатор формата всегда начинается с символа «%», может включать последовательность необязательных спецификаций и всегда завершается спецификацией преобразования. Символ «%» и спецификация преобразования являются обязательными элементами спецификатора формата. Между этими двумя обязательными элементами могут быть добавлены следующие необязательные спецификации: флаг, спецификация ширины и спецификация точности. Если необязательные спецификации присутствуют, то они должны быть записаны не в произвольном, а в приведенном выше порядке.

*Спецификация преобразования* специфицирует тип данных, которые должны быть преобразованы в строку и выведены в поле вывода. На рис. 5.8 приведены примеры спецификаторов формата, состоящие только из символа «%» и спецификации преобразования.

При выводе целочисленных данных с использованием спецификатора формата, который включает только спецификацию преобразования «d», данное выводится в поле, состоящее из такого количества позиций, которое требуется для записи числа. Например, число 12 будет выведено в поле, состоящее из двух позиции, а число –156 будет выведено в поле, состоящее из четырех позиций. При выводе чисел с плавающей запятой с использованием спецификатора формата, который включает только спецификацию преобразования «f», в дробной части числа записывается шесть значащих цифр.

| Спецификатор формата | Описание спецификатора |
|---|---|
| %d | Специфицирует вывод целочисленных данных (типы byte, short, int, long) в десятичной системе счисления |
| %f | Специфицирует вывод числовых данных с плавающей запятой (типы float, double) в десятичной системе счисления в стандартной форме |
| %e | Специфицирует вывод числовых данных с плавающей запятой (типы float, double) в десятичной системе счисления в форме «e» |
| %b | Специфицирует вывод булевых данных (тип boolean) |
| %c или %C | Специфицирует вывод символьных данных (тип char) в нижнем (%c) или верхнем (%C) регистрах |
| %s или %S | Специфицирует вывод строковых данных (тип String) в нижнем (%s) или верхнем (%S) регистрах |
| %n | Специфицирует перевод курсора на новую строку |

**Рис. 5.8.** Примеры спецификаторов формата, состоящих из символа «%» и спецификации преобразования

В спецификатор формата могут быть включены необязательные *спецификации ширины и точности*, позволяющие задать общее количество позиций поля вывода (спецификация ширины) и количество значащих цифр в дробной части числа (спецификация точности). Спецификации ширины и точности записываются в спецификаторе формата в виде целых положительных чисел. Если в спецификатор формата включена спецификация точности, то перед спецификацией точности записывается символ «точка». Спецификация ширины предполагает правое выравнивание данного в пределах поля вывода. *Под выравниванием понимается сдвиг данного в пределах поля вывода к правой или левой границе поля*. На рис. 5.9 приведены примеры спецификаторов формата, включающих спецификации ширины и точности.

| Спецификатор формата | Описание спецификатора |
|---|---|
| %<Ш>d | Специфицирует вывод целочисленных данных в десятичной системе счисления в поле вывода, состоящее из Ш позиций. При выводе осуществляется правое выравнивание. |
| %<Ш>.<Т>f | Специфицирует вывод числовых данных с плавающей запятой в десятичной системе счисления в стандартной форме в поле вывода, состоящее из Ш позиций с Т значащими цифрами в дробной части. При выводе осуществляется правое выравнивание |

**Рис. 5.9.** Примеры спецификаторов формата, включающих спецификации ширины и точности

Спецификация под наименованием «*флаг*» используется в тех случаях, когда способ размещения данного в поле вывода, описанного остальными спецификациями и принятый по правилу умолчания, необходимо изменить. Например, в том случае, когда общее количество позиций поля вывода, задаваемое спецификацией ширины больше, чем количество позиций выводимого целочисленного данного, то по правилу умолчания осуществляется правое выравнивание. При помощи символа-флага можно отменить правое выравнивание и заменить его на левое выравнивание. В таблице на рис. 5.10 приведены некоторые символы-флаги.

| Символ-флаг | Назначение флага |
|---|---|
| - | В поле вывода осуществляется левое выравнивание (по умолчанию осуществляется правое выравнивание) |
| пробел | Поле вывода начинается с символов «пробел». Количество пробелов в поле вывода равно количеству символов «пробел», записанных в спецификаторе формата |
| + | В поле вывода перед положительным числом записывается знак «+» (по умолчанию перед положительным числом знак «+» не записывается) |

**Рис. 5.10.** Символы-флаги, используемые в спецификаторах формата

Фрагмент кода на рис. 5.11 иллюстрирует применение спецификаторов формата для кодирования форматированного вывода на консоль.

Во всех случаях вывода в форматной строке последним записан спецификатор формата «%n», предписывающий перевести курсор к началу следующей строки.

В нижней части рис. 5.11 приведены строки, отображаемые на экране консоли в результате выполнения кода.

В строки 1–3 выводится значение одной и той же целочисленной переменной с именем num1. При объявлении этой переменной ей присвоено начальное значение 58. Размещение значение num1 в поле вывода специфицировано различным

образом. Поле вывода строки 1 описано спецификатором формата «%d», который определяет количество позиций поля для вывода целого числа, равное количеству разрядов этого числа. Поле вывода строки 2 описано спецификатором «%8d», включающим спецификацию ширины в виде числа 8. Значение переменной num1 выводится в поле, имеющее 8 позиций, и, поскольку в спецификаторе формата отсутствует флаг левого выравнивания, то осуществляется правое выравнивание, принятое по умолчанию. Поле вывода строки 3 описано спецификатором «%-8d», отличающегося от спецификатора строки 2 наличием флага выравнивания (символ «-»), отменяющего выравнивание по умолчанию и предписывающего левое выравнивание.

В строки 4–9 выводится значение числовой переменной с плавающей запятой, с именем num2. При объявлении переменной num2 ей присвоено начальное значение 343.75. Поле вывода строки 4 описано спецификатором «%f», который, по умолчанию, выделяет шесть позиций для дробной части числа. В спецификатор формата поля вывода строки 5, записанного в виде «%.2f», не включена спецификация ширины, но включена спецификация точности, предписывающая выделить для дробной части числа две позиции. Спецификатор предполагает также правое выравнивание по умолчанию. В спецификатор формата поля вывода строки 6, который записан в виде «%8.4f», включены и спецификация ширины, и спецификация точности, предписывающие выделить для всего поля вывода 8 позиций, а для дробной части числа — четыре позиции. Поле вывода строки 7 описано спецификатором «%-8.2f», который включает: флаг левого выравнивания, спецификацию ширины и спецификацию точности. Этот спецификатор предписывает вывести число в поле, состоящее из 8 позиций, выделить для дробной части 2 позиции и выполнить левое выравнивание. Поле вывода строки 8 описано спецификатором «%.0f», который, кроме обязательной спецификации преобразования (символ «f»), содержит спецификацию точности в виде «.0». Спецификация точности предписывает не выделять позиций для дробной части числа. Поэтому в восьмой строке в поле вывода выводится целая часть значения переменной num2 после его округления. Поле вывода строки 9 описано спецификатором «%.e», означает, что число с плавающей запятой необходимо вывести в форме «e». Как видно на рис. 5.11 в строку 9 выводится мантисса (число 3.437500), затем символ «e» и порядок со знаком (+2). По умолчанию число в форме «e» выводится с 6 позициями в дробной части.

В строки 10–12 выводятся значения переменной строкового типа с именем s и значением "форматированный вывод". Спецификатор поля вывода строки 10 записан в виде «%s» и предписывает вывести строку символов в нижнем регистре, выделив для размещения строки количество позиций, равное количеству символов в строке. В спецификатор поля вывода строки 11, записанный в виде «%25s», включена спецификация ширины, которая предписывает вывести строку в поле, состоящее из 25 позиций с выравниванием по умолчанию. Как и в случае вывода числовых данных, при выводе строковых данных выравнивание по умолчанию означает сдвиг строки к правой границе поля вывода. Спецификатор поля вывода строки 12, записанный в виде «%-25s», отличается от спецификатора поля вывода строки 11 только

```
int num1 = 58;
double num2 = 343.75;
String s = "Форматированный вывод";
char ch = 'a';
// Форматирование целочисленных данных
System.out.printf("Строка 1 = %d %n", num1);
System.out.printf("Строка 2 = %8d метров %n", num1);
System.out.printf("Строка 3 = %-8d метров %n", num1);
// Форматирование числовых данных с плавающей запятой
System.out.printf("Строка 4 = %f %n", num2);
System.out.printf("Строка 5 = %.2f метров %n", num2);
System.out.printf("Строка 6 = %8.4f метров %n", num2);
System.out.printf("Строка 7 = %-8.2f метров %n", num2);
System.out.printf("Строка 8 = %.0f метров %n", num2);
System.out.printf("Строка 9 = %e метров %n", num2);
// Форматирование строковых данных
System.out.printf("Строка 10 = %s %n", s);
System.out.printf("Строка 11 = %25s %n", s);
System.out.printf("Строка 12 = %-25s %n", s);
// Форматирование символьных данных
System.out.printf("Строка 13 = %c %n", ch);
// Форматирование нескольких данных
int a = 5, b = 10, c = 15;
System.out.printf("Строка 14 = %d % d % d %n", a, b, c);
int x = 25;
double y = 5.9;
double sum = x + y;
System.out.printf("Строка 15 = %d + %f = %.2f %n", x, y, sum);
Строка 1 = 58
Строка 2 =       58 метров
Строка 3 = 58       метров
Строка 4 = 343.750000
Строка 5 = 343.75 метров
Строка 6 = 343.7500 метров
Строка 7 = 343.75   метров
Строка 8 = 344 метров
Строка 9 = 3.437500e+02 метров
Строка 10 = форматированный вывод
Строка 11 =     форматированный вывод
Строка 12 = форматированный вывод
Строка 13 = a
Строка 14 = 5 10 15
Строка 15 = 25 + 5.900000 = 30.90
```

**Рис. 5.11.** Фрагмент кода, иллюстрирующий применение спецификаторов формата для кодирования форматированного вывода на консоль.

наличием символа флага левого выравнивания (символ «-»). Этот флаг отменяет выравнивание по умолчанию и предписывает левое выравнивание, означающее, что строка должна быть сдвинута к левой границе поля вывода.

В строку 13 выводится значение символьной переменной с именем `ch`. Спецификатор поля вывода этой переменной записан в виде «`%c`» и означает, что для вывода необходимо выделить одну позицию и вывести символ в нижнем регистре.

В строки 14 и 15 выводятся значения нескольких переменных. В строку 14 выводятся значения трех целочисленных переменных с именами `a`, `b` и `c`, которым присвоены начальные значения 5, 10 и 15, соответственно. Форматная строка, форматирующая вывод, содержит три последовательно расположенных, спецификатора формата «`%d`», «`% d`» и «`% d`», специфицирующих поля вывода переменных `a`, `b` и `c`. Все три спецификатора предписывают выделить для вывода чисел количество позиций, равное количеству цифр в числе (символ «`d`»). Однако второй и третий спецификаторы включают символ флага в виде одного символа пробела, что предписывает вначале вывести в поле вывода один пробел, а затем значение соответствующей целочисленной переменной. В строку 15 последовательно выводятся значения разнотипных переменных `x` (тип `int`) и `y` (тип `double`), а затем значение переменной `sum` (тип `double`), являющееся их суммой. Поле вывода значения переменной `x` описано спецификатором формата «`%d`», который предписывает вывести целое число и выделить для этого количество позиций, равное количеству цифр в числе. Поле вывода значения переменной `y` описано спецификатором «`%f`», который предписывает выделить для вывода дробной части числа шесть позиций. Поле вывода значения переменной `sum` описано спецификатором «`%.2f`», который содержит спецификацию точности и предписывает выделить для вывода дробной части две позиции.

## 5.3. Ввод данных с консоли

При вводе данных с консоли внешним источником данных является клавиатура консоли. В языке программирования Java существует большое количество предопределенных классов, содержащих методы, которые можно использовать для организации ввода данных с клавиатуры консоли. При изучении императивного программирования и получения навыков кодирования удобно использовать методы, объявленные в предопределенном классе `Scanner`.

Для организации ввода данных с клавиатуры консоли при помощи методов класса `Scanner` необходимо:

1. создать объект класса `Scanner`, методы которого могут считывать данные с клавиатуры консоли;
2. знать назначение и особенности работы каждого из методов класса `Scanner`, а также структуру сообщения, вызывающего требуемый метод.

### 5.3.1. Создание объекта класса Scanner

Предопределенный класс System, кроме статического поля out, включает также статическое поле in, предназначенное для хранения ссылки на объект, моделирующий клавиатуру консоли. Таким образом, статическое поле System.in можно рассматривать как имя объекта, моделирующего клавиатуру консоли. Статическое поле in объявлено в классе System при помощи следующего предложения

```
public static final InputStream in;
```

и хранит ссылку на объект класса InputStream, а не на объект класса Scanner.

Поэтому для того, чтобы получить возможность вызывать методы, объявленные в классе Scanner, необходимо создать объект этого класса, указав при его создании, что методы класса Scanner должны считывать данные с клавиатуры консоли. Предложение, при помощи которого создаются объекты класса Scanner, методы которого ориентированы на работу с клавиатурой консоли, имеет вид

```
Scanner scan = new Scanner(System.in);
```

В результате выполнения этого предложения будет создан объект класса Scanner, а ссылка на этот объект помещена в ссылочную переменную с именем scan. Имя ссылочной переменной выбирается программистом, и вместо имени scan можно использовать любое другое корректное имя, например, in.

### 5.3.2. Методы класса Scanner

В классе Scanner объявлено много методов, которые могут использоваться при организации ввода данных с клавиатуры консоли. Как правило, при изучении императивного программирования достаточными являются девять методов со следующими заголовками.

```
public byte nextByte()
public short nextShort()
public int nextInt()
public long nextLong()
public float nextFloat()
public double nextDouble()
public boolean nextBoolean()
public byte nextByte()
public String nextLine()
```

Имена перечисленных методов начинаются со слова next (следующий), поэтому в дальнейшем будем называть их next-методами. Каждый из методов специализирован на считывание с клавиатуры консоли последовательности символов (лексемы) и преобразование считанной лексемы в данное определенного типа. Лексема, считанная с клавиатуры, преобразуется в данное того типа, которое указано во второй части имени next-метода. Например, next-метод с именем nextInt преобразует считанную последовательность символов в числовое данное типа int. Преобразованное данное является возвращаемым значением next-метода. Фрагмент кода на рис. 5.12 иллюстрирует использование next-методов для кодирования ввода данных с консоли.

```
Scanner scan = new Scanner(System.in); // предложение 1

int age;                               // предложение 2
float weight;                          // предложение 3
String name;                           // предложение 4

age = scan.nextInt();                  // предложение 5
weight = scan.nextFloat();             // предложение 6
name = scan.nextLine();                // предложение 7
```

**Рис. 5.12.** Фрагмент кода, иллюстрирующий применение next-методов для кодирования ввода данных с консоли

При помощи первого предложения в коде на рис. 5.12 создается объект класса Scanner. Ссылка на этот объект помещается в переменную с именем scan. Предложения 2, 3 и 4 объявляют три переменные с именами: age (тип int); weight (тип float); name (тип String). Предложение 5 построено на основе оператора присваивания. В правой части предложения записано сообщение, вызывающее метод nextInt объекта scan. Метод nextInt считывает лексему, введенную с клавиатуры консоли, преобразует эту лексему в числовое данное типа int и возвращает полученное значение. Возвращаемое значение метода nextInt записывается в переменную age. Предложения 6 и 7 кодируют те же действия, что и предложение 5, но предназначены для считывания с клавиатуры консоли числового данного с плавающей запятой (предложение 6) и строкового данного (предложение 7). Считывание этих данных осуществляется методами nextFloat и nextLine соответственно.

### 5.3.3. Диалоговый режим ввода данных с консоли

Ввод данных с консоли, как правило, осуществляется в режиме диалогового взаимодействия программы с ее пользователем. Диалоговое взаимодействие может быть представлено в виде последовательности транзакций. Отдельная *диалоговая*

*транзакция* состоит из двух частей: (1) вывода на экран консоли запроса о том, какие данные необходимо ввести, и (2) ввода требуемых данных пользователем при помощи клавиатуры консоли.

Выполнение кода, приведенного на рис. 5.12, может поставить пользователя в тупиковую ситуацию, когда он не будет знать, что от него ожидает программа. Для исключения таких ситуаций при кодировании ввода с консоли целесообразно предварять ввод данных выводом на экран консоли запроса, подсказывающего пользователю, какие именно данные он должен ввести. На рис. 5.13 приведен фрагмент кода, иллюстрирующего диалоговый режим ввода данных с консоли. Код на рис. 5.13 получен из кода на рис. 5.12 путем добавления предложений, организующих диалоговый режим ввода данных. В нижней части рисунка приведены диалоговые транзакции, отображаемые на экране консоли.

```
Scanner scan = new Scanner(System.in);

int age;
doble weight;
String name;

System.out.println("Введите свой возраст и нажмите Enter"); // Код первой
age = scan.nextInt();                                       // транзакции

System.out.println("Введите свой вес и нажмите Enter");     // Код второй
weight = scan.nextDouble();                                 // транзакции

System.out.println("Введите свое имя и нажмите Enter");     // Код третьей
name = scan.nextLine();                                     // транзакции


Введите свой возраст и нажмите Enter  } первая транзакция
5
Введите свой вес и нажмите Enter      } вторая транзакция
50.8
Введите свое имя и нажмите Enter      } третья транзакция
Дмитрий
```

**Рис. 5.13.** Фрагмент кода, иллюстрирующий диалоговый режим ввода данных с консоли

Часть кода, осуществляющая ввод данных, организована в виде последовательности, состоящей из трех транзакций. Каждая транзакция кодируется двумя предложениями. Первое предложение выводит на экран консоли запрос на ввод данного, а второе — осуществляет ввод данного с клавиатуры консоли. В коде на рис. 5.13

для кодирования первого предложения (вывода запроса на экран консоли) используется метод `println` предопределенного класса `PrintStream`, а для кодирования второго предложения (ввода данных с клавиатуры консоли) — соответствующий `next`-метод класса `Scanner`.

## Упражнения для самостоятельной работы

5.1. Разработайте программный код, который считывает с консоли три целых числа и выводит на экран их среднее арифметическое значение. Результат округлите до 5 значащих цифр в дробной части.

5.2. Разработайте программный код, который считывает с консоли два числа с плавающей запятой и выводит на экран их сумму, разность и произведение.

5.3. Разработайте программный код, который запрашивает и считывает с консоли целое число, равное длине стороны квадрата, а затем выводит на экран периметр и площадь квадрата.

5.4. Разработайте программный код, который преобразует температуру по шкале Фаренгейта в температуру по шкале Цельсия. Значение температуры по шкале Фаренгейта вводится пользователем с клавиатуры консоли.

5.5. Разработайте программный код, который преобразует мили в километры. (Одна миля равна 1,60935 километров.) Длина в милях вводится пользователем с клавиатуры консоли в виде значения с плавающей запятой.

5.6. Разработайте программный код, который считывает с консоли текущее показание электронных часов: $h$ часов ($0 \le h \le 23$), $m$ мин ($0 \le m \le 59$), $s$ сек ($0 \le s \le 59$) и выводит на экран общее количество секунд, прошедших от начала суток, например, в следующем виде: «1 час, 28 минут и 42 секунды эквивалентно 5 322 секундам».

5.7. Измените программный код упражнения 5.6 таким образом, чтобы с консоли считывалось значение, представляющее собой количество секунд, прошедших от начала суток, а на экран выводилось эквивалентное количество времени в виде комбинации часов, минут и секунд, например, в следующем виде: «9999 секунд эквивалентны 2 часам, 46 минутам и 39 секундам».

5.8. Разработайте программный код, который запрашивает и считывает с консоли фамилию, имя и отчество студента, а выводит на экран только фамилию и инициалы.

5.9. Разработайте программный код, который запрашивает и считывает с консоли целое четырехразрядное число, а затем выводит на экран сумму его цифр.

5.10. Разработайте программный код, который запрашивает и считывает с консоли целое трехразрядное число, а затем выводит на экран число, которое получается из введенного числа путем перестановки цифры в разряде сотен с цифрой в разряде единиц.

5.11. Разработайте программный код, который запрашивает и считывает с консоли положительное целое четырехразрядное число, а затем выводит на экран число, которое получается из введенного числа путем перестановки первых двух и последних двух цифр.

5.12. Пусть три резистора $R_1$, $R_2$, $R_3$ соединены параллельно. Разработайте программный код, который запрашивает и считывает с консоли значения сопротивлений этих резисторов и выводит на экран значение общего сопротивления соединения. Результат округлите до 4 значащих цифр в дробной части.

Раздел 6

# ПРОГРАММИРОВАНИЕ ВЕТВЛЕНИЙ

Программный код состоит из предложений, сгруппированных в логически целостные части, или блоки. Предложения внутри блока записываются в том порядке, в котором они должны выполняться компьютером. Иными словами, предполагается, что после выполнения какого-либо из предложений выполняется предложение, расположенное непосредственно за ним. Такой способ кодирования называют *кодированием с естественным порядком следования предложений*. Если программный код состоит только из предложений, подчиняющихся естественному порядку следования, то программа имеет линейную структуру и представляет собой цепь последовательно следующих друг за другом предложений. Кодирование с естественным порядком следования предложений не является единственно возможным способом записи кода программы. Можно, например, снабдить каждое предложение уникальным идентификатором, а в конце предложения разместить ссылку на то предложение, которое должно выполняться следующим. В этом случае предложения в коде могут быть размещены в произвольном порядке.

Кодирование с естественным порядком следования предложений общепринято для языков программирования высокого уровня по нескольким причинам. Главная причина в том, что тексты таких программ легко воспринимаемы, поскольку при восприятии и анализе кода с естественным порядком следования предложений мы опираемся на знакомые навыки восприятия естественно-языковых текстов.

Однако, только очень простой программный код имеет строго линейную структуру. Как правило, код включает некоторое количество предложений, являющихся аналогами вопросительных предложений естественного языка, которые предполагают выбор одного из некоторого количества альтернативных ответов. В простейшем случае это вопросительное предложение типа «да/нет», предполагающее выбор одного из двух альтернативных ответов. Если при выполнении программного кода с естественным порядком следования предложений встречается подобное предложение, то оно является точкой, разделяющей выполнение программного кода на несколько альтернативных направлений. Достигнув точки ветвления, компьютер вначале определяет ответ, а затем выбирает и выполняет один из альтернативных блоков, соответствующий ответу. Остальные альтернативные блоки игнорируются. Например, определив, количество символов в некоторой строке, мы можем задать вопрос: «является ли это количество символов

четным числом?». Ответ на этот вопрос определяет выбор и выполнение одного из двух альтернативных блоков. Если количество символов в строке является нечетным числом, то программа может выполнить блок, выделяющий средний символ этой строки.

Для включения в код предложений, при помощи которых можно изменить естественный порядок следования предложений и организовать разветвление программного кода на несколько альтернативных направлений, в языке программирования Java имеются два оператора, обозначенные служебными словами `if` и `switch`.

## 6.1. Оператор `if`

Оператор `if` является аналогом вопросительного предложения типа «да/нет» и используется в тех случаях, когда необходимо выбрать и выполнить один их двух альтернативных блоков кода.

Существуют два типа оператора `if`: упрощенный и полный. С упрощенным оператором `if` связан только один блок, который может быть либо выполнен, либо проигнорирован. Будем такой оператор называть оператором `if` типа «одно условие — один блок». С полным оператором `if` связаны два альтернативных блока, из которых может выполниться только один. Будем такой оператор называть оператором `if` типа «одно условие — два блока». Структура кода, использующего оператор `if` типа «одно условие — один блок», приведена на рис. 6.1.

```
if(<условие ветвления>)
   {<блок>}
<последующие предложения>
```

**Рис. 6.1.** Структура кода, использующего оператор `if` типа «одно условие — один блок»

Как видно на рис. 6.1, при использовании оператора `if` типа «одно условие — один блок» код имеет следующую структуру. Вначале записывается строка, в которой после служебного слова `if` размещается условие ветвления в круглых скобках. Условие ветвления представляет собой булево выражение и может быть любым из изученных ранее булевых выражений: литерал или переменная булевого типа; булево выражение сравнения; булево выражение с логическими операциями и т. п.

После строки с условием размещаются предложения программного блока, заключенные в фигурные скобки. При выполнении кода, структура которого приведена на рис. 6.1, вначале определяется значение булевого выражения в круглых скобках. Если это значение равно `true`, то выполняется блок, размещенный непосредственно после строки с условием. После завершения выполнения предложений этого блока выполняются последующие предложения. Если значение булевого

выражения в круглых скобках равно `false`, то блок в фигурных скобках игнорируется, и управление сразу передается последующим предложениям.

Проиллюстрируем использование оператора `if` типа «одно условие — один блок» на примере кодирования следующей задачи. Пусть для некоторой строки символов необходимо выделить средний символ, но только в том случае, когда эта строка содержит нечетное количество символов. Возможное решение приведено на рис. 6.2.

```
String word = "...";                          // предложение 1
int numbOfSymbols, midIndex;                  // предложение 2
char midSymbol = '...';                       // предложение 3
   numbOfSymbols = word.length();             // предложение 4
   if(numbOfSymbols%2 != 0){                  // оператор if
      midIndex = word.length()/2;             // предложение 5
      midSymbol = word.charAt(midIndex);      // предложение 6
   }
// последующие предложения
```

**Рис. 6.2.** Пример кода с оператором `if` типа «одно условие — один блок»

Предложение 1 на рис. 6.2 создает строку с именем `word` и некоторым значением (отмеченным многоточием). Это та строка, средний символ которой необходимо выделить, если строка содержит нечетное количество символов. Предложение 2 объявляет две целочисленные переменные. Переменная с именем `numbOfSymbols` предназначена для хранения количества символов строки `word`, а переменная с именем `midIndex` — для хранения индекса среднего символа этой строки. Предложение 3 объявляет и инициализирует переменную символьного типа с именем `midSymbol`, предназначенную для хранения значения среднего символа. Сообщение в правой части предложения 4 вызывает метод `length` строки `word`, возвращающий длину этой строки, которая присваивается переменной `numbOfSymbols`. Дальнейшая работа программы зависит от того, является ли значение переменной `numbOfSymbols` числом четным или нечетным. Проверка этого факта осуществляется при помощи булевого выражения `numbOfSymbols%2 != 0`. Это булево выражение сравнения, которое принимает значение `true` в том случае, когда выражение `numbOfSymbols%2` возвращает число, отличное от нуля. Выражение `numbOfSymbols%2` возвращает остаток от деления значения переменной `numbOfSymbols` на два. Ясно, что если в переменной `numbOfSymbols` находится четное число, то этот остаток равен нулю и булево выражение принимает значение `false`, а если в переменной `numbOfSymbols` находится нечетное число, то остаток от деления на два не равен нулю и булево выражение принимает значение `true`. Блок в фигурных скобках, состоящий из предложений 5 и 6, выполняется только в том случае, когда булево выражение в операторе `if` принимает значение `true`, что, как мы только что выяснили, происходит в том случае, когда количество символов в строке `numbOfSymbols` является нечетным.

Выделение среднего символа строки с использованием методов `length` и `charAt` мы рассматривали ранее (см. рис. 4.11).

На рис. 6.3 представлена структура программного кода в случае использования оператора `if` типа «одно условие — два блока».

```
if(<условие ветвления>)
   {<блок 1>}
else
   {<блок 2}
<последующие предложения>
```

**Рис. 6.3.** Структура кода, использующего оператор `if` типа «одно условие — два блока»

Как и в случае оператора `if` типа «одно условие — один блок», вначале записывается строка, в которой после служебного слова `if` размещается условие ветвления в круглых скобках. Непосредственно после этой строки в фигурных скобках размещается блок 1. Этот блок выполняется в том случае, когда условие ветвления принимает значение `true`. Затем после служебного слова `else` в фигурных скобках размещается блок 2, который выполняется, если условие ветвления принимает значение `false`. Блоки 1 и 2 альтернативны и, следовательно, выполниться может только один из них. После завершения выполнения любого из блоков 1 или 2 выполняются последующие предложения.

Проиллюстрируем использование оператора `if` типа «одно условие — два блока» на примере решения следующей задачи. Пусть необходимо создать строку-приветствие с одним из следующих альтернативных значений: `"Добро пожаловать"` или `"Пароль не действителен"`. Строка `"Добро пожаловать"` создается в том случае, когда введенный пароль совпадает с действующим паролем, а строка `"Пароль не действителен"` — в том случае, когда введенный пароль не совпадает с действующим. На рис. 6.4 приведен код, решающий сформулированную задачу.

```
String input;                                  // предложение 1
String validPassword;                          // предложение 2
String greeting;                               // предложение 3
// создание и инициализация строк input и validPassword
   if(validPassword.equals(input)){            // оператор if
      greeting = "Добро пожаловать";           // предложение 4
      }
   else {
      greeting = "Пароль не действителен";     // предложение 5
      }
// последующие предложения
```

**Рис. 6.4.** Пример кода с оператором `if` типа «одно условие — два блока»

Предложения 1–3 на рис. 5.4 объявляют ссылочные переменные типа `String`. Переменная с именем `input` предназначена для ссылки на строку, значение которой равно введенному паролю. Переменная с именем `validPassword` предназначена для ссылки на строку, значение которой равно действующему паролю. Переменная с именем `greeting` предназначена для ссылки на строку-приветствие, которая создается в результате работы кода. После создания и инициализации строк `input` и `validPassword` выполняется оператор `if(validPassword.equals(input))`. Булево выражение в этом операторе принимает значение `true` только в том случае, когда значения строк `input` и `validPassword` совпадают. В этом случае выполняется блок, состоящий из предложения 4, которое создает строку с именем `greeting` и значением `"Добро пожаловать"`. Если значения строк `input` и `validPassword` не совпадают, то булево выражение в операторе `if` принимает значение `false`, и выполняется блок, следующий непосредственно после служебного слова `else`. Этот блок состоит из предложения 5, которое создает строку с именем `greeting` и значением `"Пароль не действителен"`.

В том случае, если блок в операторе `if` состоит из одного предложения, то это предложение можно не заключать в фигурные скобки. Поэтому код, приведенный на рис. 6.4, можно переписать в виде, приведенном на рис. 6.5.

```
String input;                                  // предложение 1
String validPassword;                          // предложение 2
String greeting;                               // предложение 3
// создание и инициализация строк input и validPassword
   if(validPassword.equals(input))             // оператор if
      greeting = "Добро пожаловать";           // предложение 4
   else
      greeting = "Пароль не действителен";     // предложение 5
// последующие предложения
```

**Рис. 6.5.** Пример кода, с оператором `if` типа «одно условие — два блока». Альтернативные блоки кода не заключены в фигурные скобки

Однако, рекомендуется использовать фигурные скобки во всех случаях, поскольку это помогает предотвратить появление ошибок, когда, например, в процессе редактирования кода в блок, первоначально состоящий из одного предложения, добавляются новые предложения.

### 6.1.1. Диаграмма деятельности

Поведение программной системы, особенно в тех случаях, когда имеет место нарушение естественного порядка следования предложений, удобно моделировать графически, при помощи диаграммы деятельности унифицированного языка моделирования UML.

В диаграмме деятельности поведение рассматривается как последовательности шагов-деятельностей, приводящих к достижению одной или нескольких целей. Деятельность понимается в широком смысле. Это либо выполнение отдельного предложения, либо выполнение блока предложений, либо выполнение отдельного метода, либо выполнение группы логически связанных методов. При помощи специальных графических символов диаграмма деятельности легко моделирует как линейные блоки, так и ветвления в программном коде, организуемые при помощи оператора `if`.

На рис. 6.6 приведены две диаграммы деятельности, моделирующие поведение программы, содержащей оператор `if` типа «одно условие — один блок» (слева) и оператор `if` типа «одно условие — два блока» (справа).

Начало диаграммы деятельности обозначается *стартовым символом*, представляющим собой жирную точку, а завершение диаграммы — *символом завершения*, который изображается в виде жирной точки внутри окружности. Символ завершения соответствует некоторому целевому состоянию.

Графическим *символом деятельности* является прямоугольник с закругленными углами, а сама диаграмма деятельности представляет собой множество графических символов деятельности, связанных между собой *символами переходов* от предыдущей деятельности к последующей.

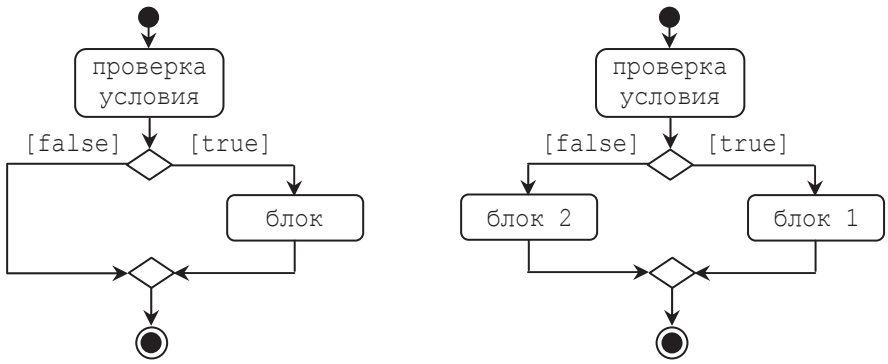



**Рис. 6.6.** Графическое представление двух типов оператора `if` в виде диаграммы деятельности

На рис. 6.6 символы деятельности использованы для моделирования деятельности по проверке истинности условия ветвления и для моделирования блока предложений. При отображении диаграммы деятельности в программный код важно знать и помнить следующее правило. *Каждый графический символ деятельности, моделирующий блок предложений, порождает пару фигурных скобок в программном коде*.

Графический символ перехода, или ветвь, изображается в виде обыкновенной стрелки, исходящей из символа предыдущей деятельности и входящей в символ последующей деятельность.

Кроме графических символов деятельности и перехода, а также символов начала и завершения, диаграмма деятельности использует графические *символы ветвления и соединения*, изображаемые в виде ромбов.

Символ ветвления имеет одну входящую ветвь и две исходящие ветви. На рис. 6.6 каждая из исходящих ветвей символа ветвления соответствует одному из значений булевого выражения-условия. Будем исходящую ветвь символа ветвления, которая соответствует значению `true`, называть `true`-ветвью, а исходящую ветвь, соответствующую значению `false`, называть `false`-ветвью. На рис. 6.6 `true`-ветвь направлена на графический символ деятельности, который моделирует работу блока, предложения которого выполняются в том случае, если булево выражение принимает значение `true`. А `false`-ветвь направлена либо на графический символ деятельности, который моделирует работу блока, который выполняется в случае, если булево выражение принимает значение `false` (тип «одно условие — два блока»), либо на символ соединения, ведущий к символу завершения диаграммы деятельности (тип «одно условие — один блок»).

Символ соединения имеет две входные ветви и одну выходную. Этот символ используется для моделирования того места на диаграмме, где завершается участок с разветвлением кода и начинается линейный участок кода. Ясно, что на диаграмме деятельности *количество символов ветвления должно быть равно количеству символов соединения*.

В ряде случаев логика разветвлений может быть весьма изощренной, и при кодировании легко совершить как синтаксические, так и логические ошибки. Моделирование поведения системы при помощи диаграммы деятельности позволяет уменьшить количество возможных ошибок.

## 6.2. Вложенные операторы `if`

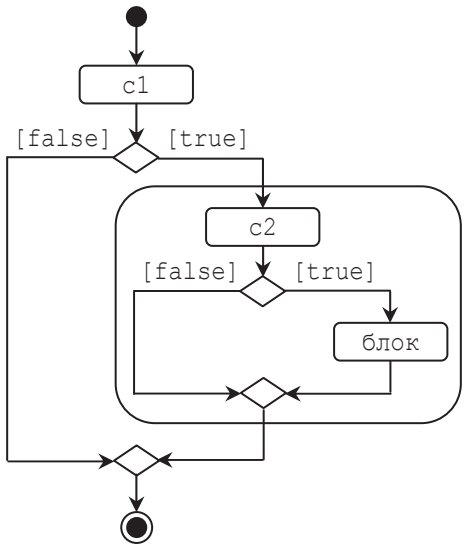
Блоки программного кода, которые используются в операторе `if`, не обязательно имеют линейную структуру и, в свою очередь, могут включать оператор `if`. Такие конструкции называются *вложенными операторами* `if`. Введем термины «внешний оператор `if`» и «внутренний оператор `if`», смысл которых очевиден. Внутренний оператор вкладывается в программный блок внешнего оператора. Во внутренний оператор `if` разрешается вкладывать еще один оператор `if` и так далее.

Существует множество структурных вариантов программного кода, образующихся при вложении одного оператора `if` в программные блоки другого оператора `if`. Эти структурные варианты определяются типами внешнего и внутреннего операторов `if`, а также тем, в какую из ветвей внешнего оператора `if` осуществляется вложение. Для того, чтобы осмысленно строить программный код с вложенными операторами `if` и не делать при этом ни логических, ни синтаксических ошибок, проанализируем возможные структурные варианты для случая, когда степень вложения равна двум (один внутренний оператор `if` вкладывается в один внешний оператор `if`). Если ограничиться степенью вложения, равной двум, то все структурные варианты можно типизировать по количеству условий ветвления и блоков, используемых тем или иным структурным вариантом. Общее количество вариантов определяется следующим списком:

- вложенный оператор `if` типа «два условия — один блок»;
- вложенный оператор `if` типа «два условия — два блока»;
- вложенный оператор `if` типа «два условия — три блока»;
- вложенный оператор `if` типа «три условия — три блока»;
- вложенный оператор `if` типа «три условия — четыре блока».

### 6.2.1. Вложенный оператор `if` типа «два условия — один блок»

Вложенный оператор `if` типа «два условия — один блок» имеет простейшую структуру и образуется в том случае, когда и внутренний, и внешний операторы `if` относятся к типу «одно условие — один блок». На рис. 6.7 приведен вариант вложенного оператора `if` типа «два условия — один блок».




```
if(c1){
   if(c2)
      {<блок>}
}
<последующие предложения>
```

**Рис. 6.7.** Вложенный оператор `if` типа «два условия — один блок».
`c1` — условие ветвления внешнего оператора; `c2` — условие ветвления внутреннего оператора

В левой части рис. 6.7 изображена модель в виде диаграммы деятельности, а в правой — соответствующая структура программного кода.

Модель на рис. 6.7 выражает в графической форме некоторое правило, регламентирующее выполнение блока. Правило связывает между собой условие и действие. Действие выполняется только в том случае, когда имеет место условие. Правила типа «условие — действие» называются продукционными. Запишем продукционное правило для вложенного оператора `if` типа, приведенного на рис. 6.7, в виде.

```
c1 && c2 ⇒ <блок>
```

Продукционное правило состоит из левой и правой частей. В качестве разделителя между левой и правой частями использован символ импликации, который читается как «влечет за собой». Левая часть правила специфицирует условие в виде булева выражения. Булево выражение состоит из условий ветвления (либо их отрицаний), связанных логической операцией «И». В терминах пропозициональной логики левая часть продукционного правила — это дизъюнкция, дизъюнктами которой являются условия ветвления либо их отрицания. Правая часть — это действие, которое должно быть выполнено, если булево выражение в левой части истинно.

Согласно приведенному продукционному правилу, блок, находящийся в `true`-ветви внутреннего оператора `if`, выполняется только в том случае, когда условие ветвления внешнего оператора `if` принимает значение `true` и условие ветвления внутреннего оператора `if` также принимает значение `true`. При всех остальных комбинациях значений внешнего и внутреннего условий блок не должен выполняться.

Продукционное правило является, по сути, разновидностью условного предложения и легко конвертируется в оператор `if` типа «одно условие — один блок». Поэтому вложенный оператор `if` типа «два условия — один блок» эквивалентен одному оператору `if` типа «одно условие — один блок», у которого условие ветвления имеет вид `c1 && c2`. На рис. 6.8 приведена модель и структура программного кода, эквивалентные модели и коду, изображенных на рис 6.7.

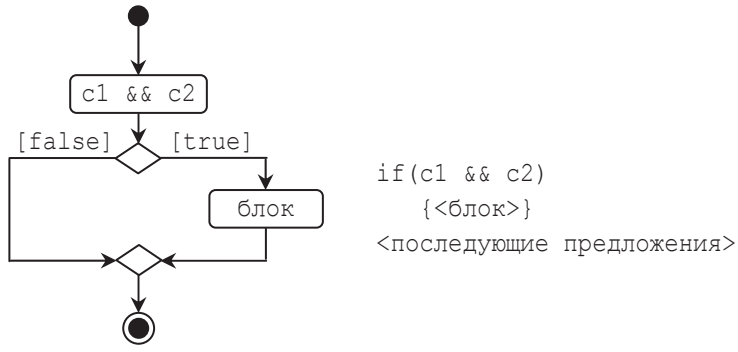



**Рис. 6.8.** Представление вложенного оператора `if` типа «два условия — один блок» одним оператором типа «одно условие — один блок»

Проиллюстрируем использование вложенного оператора `if` типа «два условия — один блок» на примере решения следующей задачи. Пусть имеется несколько терминалов, обеспечивающих доступ к информационным ресурсам. Пользователю разрешен доступ к ресурсам только через один, конкретный терминал с номером 100, при условии, что с этого терминала он введет без ошибок свой действующий идентификационный номер. Необходимо написать код, который создает строку-сообщение со значением `"Доступ разрешен"`, если идентификационный номер введен с терминала 100 и не содержит ошибок. На рис. 6.9 приведен код, решающий сформулированную задачу при помощи вложенного оператора `if` типа «два условия — один блок», приведенного на рис. 6.7.

Предложение 1 на рис. 6.9 объявляет целочисленную переменную с именем `numbOfTerminal`. Значение этой переменной соответствует номеру терминала, с которого осуществляется попытка доступа. Предложения 2–4 объявляют ссылочные переменные типа `String`. Переменная с именем `input` предназначена для ссылки на строку, значение которой соответствует *введенному идентификационному номеру*. Переменная с именем `idNumb` предназначена для ссылки на строку, значение которой соответствует *действующему идентификационному номеру*. Переменная с именем `message` предназначена для хранения ссылки на строку, которая создается при работе рассматриваемого кода.

```
int numbOfTerminal;                          // предложение 1
String input;                                // предложение 2
String idNumb;                               // предложение 3
String message;                              // предложение 4
// инициализация numbOfTerminal, создание и инициализация input и idNumb
   if(numbOfTerminal = 100)                  // внешний оператор if
      {if(idNumb.equals(input))              // внутренний оператор if
         {message = "Доступ разрешен";}}     // предложение 5
// последующие предложения
```

**Рис. 6.9.** Код, иллюстрирующий использование вложенного оператора `if` типа «два условия — один блок»

Булево выражение во внешнем операторе `if(numbOfTerminal = 100)` принимает значение `true` в том случае, когда доступ осуществляется с терминала номер 100. Булево выражение во внутреннем операторе `if(idNumb.equals(input))` принимает значение `true` в том случае, когда введенный идентификационный номер эквивалентен действующему идентификационному номеру. Блок, состоящий из одного предложения 5, выполняется, если оба отмеченных булевых выражения принимают значение `true`. Если булево выражение внешнего или внутреннего оператора `if` принимает значение `false`, то выполняются последующие предложения.

На рис. 6.10 приведен код, решающий ту же задачу, но при помощи оператора типа «одно условие — один блок».

```
// инициализация numbOfTerminal, создание и инициализация input и idNumb
   if(numbOfTerminal = 100)&&(idNumb.equals(input))
         {message = "Доступ разрешен";}
// последующие предложения
```

**Рис. 6.10.** Код, иллюстрирующий использование оператора `if` типа «одно условие — один блок» вместо вложенного оператора типа «два условия — один блок»

### 6.2.2. Вложенные операторы if типа «два условия — два блока»

Возможны три варианта вложенных операторов `if` типа «два условия — два блока». На рис. 6.11 приведен первый вариант, который образуется, если в блок `true`-ветви внешнего оператора `if` типа «одно условие — один блок» вложить внутренний оператор `if` типа «одно условие — два блока».

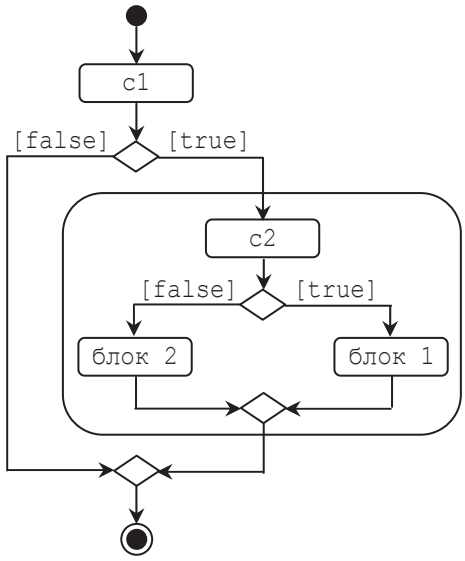
Поскольку модель на рис. 6.11 содержит два блока, то необходимы два продукционных правила, которые регламентируют их выполнение.

```
c1 && c2 ⇒ <блок1>
c1 && (!c2) ⇒ <блок2>
```

Словесно эти правила можно описать следующим образом.

Блок 1 должен выполняться в том случае, когда условие ветвления внешнего оператора `if` принимает значение `true` и условие ветвления внутреннего оператора `if` также принимает значение `true`.

Блок 2 должен выполняться в том случае, когда условие ветвления внешнего оператора `if` принимает значение `true`, а условие ветвления внутреннего оператора `if` — значение `false`. Если условие ветвления внешнего оператора `if` принимает значение `false`, то ни один из блоков не должен выполняться.

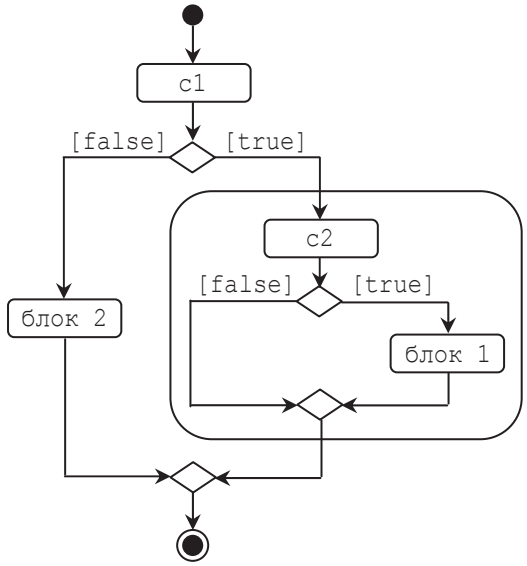



```
if(c1){
   if(c2){
      {<блок 1>}
    else
      {<блок 2>}
   }
}
<последующие предложения>
```

**Рис. 6.11.** Вложенный оператор `if` типа «два условия — два блока», вариант 1.
`c1` — условие ветвления внешнего оператора; `c2` — условие ветвления внутреннего оператора

На рис. 6.12 приведен второй вариант вложенного оператора `if` типа «два условия — два блока». Вариант соответствует случаю, когда в блок `true`-ветви внешнего

оператора `if` типа «одно условие — два блока» вкладывается внутренний оператор `if` типа «одно условие — один блок».




```
if(c1){
   if(c2)
      {<блок 1>}
   }
else
   {<блок 2>}
<последующие предложения>
```

**Рис. 6.12.** Вложенный оператор `if` типа «два условия — два блока», вариант 2. `c1` — условие ветвления внешнего оператора; `c2` — условие ветвления внутреннего оператора

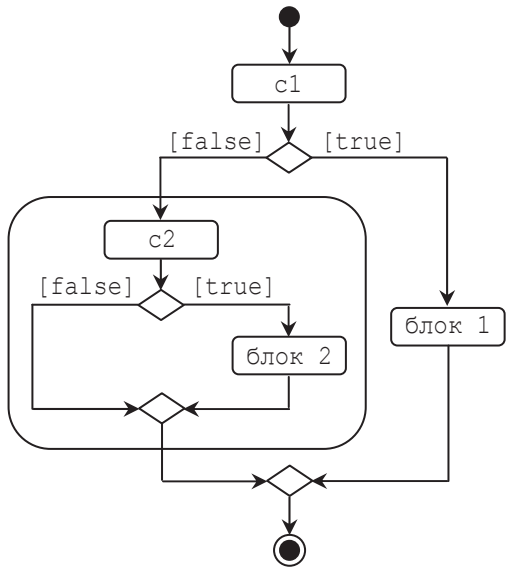
Продукционные правила, регламентирующие выполнение блоков для вложенного оператора `if` на рис. 6.12, имеют вид.

```
c1 && c2 ⇒ <блок1>
   !c1 ⇒ <блок2>
```

Блок 1 выполняется только в том случае, когда условие ветвления внешнего оператора `if` принимает значение `true` и условие ветвления внутреннего оператора `if` также принимает значение `true`. Блок 2 выполняется только в том случае, когда условие ветвления внешнего оператора `if` принимает значение `false`.

В правой части рис. 6.12 приведена структура программного кода, соответствующая модели. Эта структура очень похожа на структуру программного кода, приведенную на рис. 6.11. *Отличие заключается только в месте расположения фигурных скобок*. При кодировании вложенных операторов `if` легко допустить логическую ошибку при выделении программных блоков с помощью фигурных скобок. Предварительное представление вложенных операторов `if` в виде модели позволяет уменьшить риск такой ошибки.

На рис. 6.13 приведен третий вариант вложенного оператора `if` типа «два условия — два блока». Этот вариант образуется, если в блок `false`-ветвь внешнего оператора `if` типа «одно условие — два блока» вкладывается внутренний оператор `if` типа «одно условие — один блок».




```
if(c1)
   {<блок 1>}
else {
   if(c2)
      {<блок 2>}
}
<последующие предложения>
```

**Рис. 6.13.** Вложенный оператор `if` типа «два условия — два блока», вариант 3. `c1` — условие ветвления внешнего оператора; `c2` — условие ветвления внутреннего оператора

Продукционные правила, соответствующие вложенному оператору `if` на рис. 6.13, имеют вид.

```
c1 ⇒ <блок1>
(!c1) && c2 ⇒ <блок2>
```

Блок 1 выполняется только в том случае, когда условие ветвления внешнего оператора `if` принимает значение `true`. Блок 2 выполняется, когда условие ветвления внешнего оператора `if` принимает значение `false`, а условие ветвления внутреннего оператора `if` — значение `true`.

Все три варианта вложенного оператора `if` типа «два условия — два блока» решают одну и ту же задачу императивного программирования — *выбор и выполнение одного из двух альтернативных блоков в тех случаях, когда выбор детерминируется двумя независимыми условиями*. Таким образом, если программист сталкивается с такой задачей, он может использовать любой из трех вариантов.

Проиллюстрируем справедливость приведенного утверждения на примере решения следующей задачи. Пусть имеется резервуар для хранения запаса воды в водонапорной башне. Вода из резервуара непрерывно подается потребителям, а ее убыль компенсируется при помощи насоса, который закачивает воду в резервуар из близлежащего водоема. Имеется система контроля, которая измеряет фактический уровень воды и информирует диспетчера об уровне воды в резервуаре при помощи одного из двух альтернативных сообщений. Строка с сообщением: "`Уровень выше верхнего допустимого`" формируется, если текущий уровень воды в резервуаре

превышает верхний допустимый, а строка: "Уровень в норме", если текущий уровень находится между нижним и верхним допустимыми уровнями.

На рис. 6.14 приведены примеры кодов, решающих сформулированную задачу на основе рассмотренных вариантов вложенного оператора if типа «два условия — два блока».

```
float lowerLevel,
     upperLevel,
     currentLevel;                          // предложение 1
String message;                             // предложение 2
// инициализация lowerLevel, upperLevel и актуализация currentLevel
if(currentLevel > lowerLevel){              // внешний оператор if
   if(currentLevel > upperLevel)            // внутренний оператор if
     {message = "Уровень выше верхнего допустимого";}
   else
     {message = "Уровень в норме";}
}
// последующие предложения
```

а)

```
if(currentLevel < upperLevel){              // внешний оператор if
   if(currentLevel > lowerLevel)            // внутренний оператор if
      {message = "Уровень в норме";}
}
else
   {message = "Уровень выше верхнего допустимого";}
// последующие предложения
```

б)

```
if(currentLevel > upperLevel)               // внешний оператор if
   {message = "Уровень выше верхнего допустимого";}
else{
   if(currentLevel > lowerLevel)            // внутренний оператор if
      {message = "Уровень в норме";}
}
// последующие предложения
```

в)

**Рис. 6.14.** Решение задачи о водонапорной башне с использованием различных вариантов вложенного оператора if типа «два условия — два блока».
а) соответствует варианту 1 на рис. 6.11;
б) соответствует варианту 2 на рис. 6.12;
в) соответствует варианту 3 на рис. 6.13

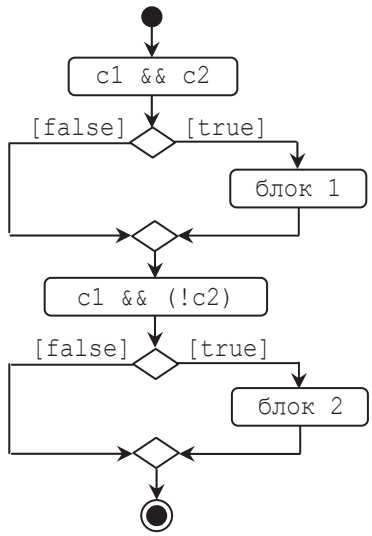
Предложение 1 на рис. 6.14 объявляет три переменные для хранения значений допустимых нижнего и верхнего уровней воды в резервуаре, а также текущего уровня воды: `lowerLevel` (нижний уровень), `upperLevel` (верхний уровень) и `currentLevel` (текущий уровень). Предложение 2 объявляет ссылочную переменную с именем `message` типа `String`. Эта переменная в дальнейшем будет ссылаться на строку, при помощи которой диспетчер информируется об уровне воды в резервуаре. Последующие части кодов моделируют рассуждения, в результате которых формируется одна из строк, и построены на основе вариантов вложенного оператора `if` типа «два условия — два блока».

На рис. 6.14 а) сообщения формируются на основе следующих рассуждений. Вначале проверяется, не является ли текущий уровень выше нижнего допустимого. Если текущий уровень выше нижнего допустимого, то осуществляется вторая проверка и проверяется, не является ли текущий уровень выше верхнего допустимого. Если выясняется, что текущий уровень выше верхнего допустимого, то формируется сообщение `"Уровень выше верхнего допустимого"`. Если из второй проверки следует, что текущий уровень ниже верхнего допустимого, то формируется сообщение `"Уровень в норме"`.

На рис. 6.14 б) сообщения формируются на основе следующих рассуждений. Вначале проверяется, не является ли текущий уровень ниже верхнего допустимого. Если текущий уровень ниже верхнего допустимого, то выполняется вторая проверка и проверяется, не является ли текущий уровень выше нижнего допустимого. Если из второй проверки следует, что текущий уровень выше нижнего допустимого, то формируется сообщение `"Уровень в норме"`. Если из первой проверки следует, что текущий уровень выше верхнего допустимого, то формируется сообщение `"Уровень выше верхнего допустимого"`.

На рис. 6.14 в) строки с сообщениями формируются на основе следующих рассуждений. Вначале проверяется, не является ли текущий уровень выше верхнего допустимого. Если это так, то формируется сообщение `"Уровень выше верхнего допустимого"`. Если это не так, то выполняется вторая проверка и проверяется, не является ли текущий уровень выше нижнего допустимого. Если из второй проверки следует, что текущий уровень выше нижнего допустимого, то формируется сообщение `"Уровень в норме"`.

Как уже было отмечено, структурные варианты вложенного оператора `if` типа «два условия — два действия» решают задачу выбора и выполнения одного из двух альтернативных блоков в случае, когда выбор детерминируется двумя независимыми условиями. Однако эту задачу не обязательно решать при помощи вложенного оператора `if`. Любой из вариантов вложенного оператора `if` типа «два условия — два блока» можно заменить на два последовательно расположенных оператора `if` типа «одно условие — один блок», у которых, в качестве условий ветвления, записаны условия, сформулированные в левых частях соответствующих продукционных правил. На рис. 6.15 приведена модель и структура программного кода, логически эквивалентные вложенному оператору `if` на рис. 6.11.




```
if(c1 && c2)
   {<блок 1}
if(c1 && (!c2))
   {<блок 2}
```

**Рис. 6.15.** Два последовательно расположенных оператора `if`типа «одно условие — один блок» вместо вложенного оператора `if` типа «два условия — два блока»

Проиллюстрируем применимость модели на рис. 6.15 для решения задачи о водонапорной башне. Пусть

`c1` соответствует `currentLevel > lowerLevel`
`c2` соответствует `currentLevel > upperLevel`

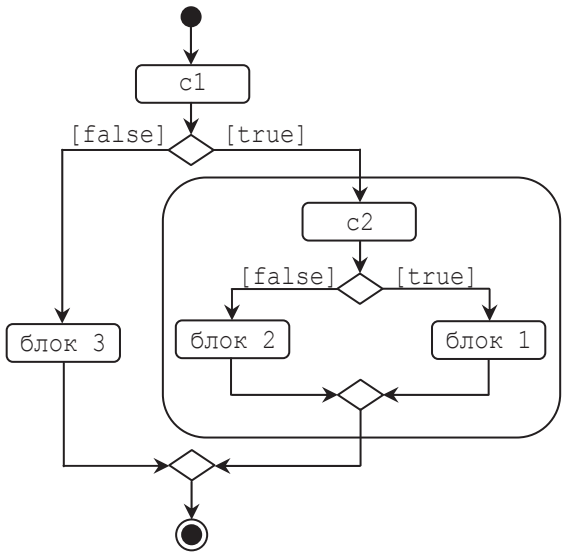
тогда фрагмент программного кода, решающего задачу о водонапорной башне, в соответствии с моделью на рис 6.15 можно представить рис. 6.16.

```
if((currentLevel > lowerLevel) && (currentLevel > upperLevel))
   {message = "Уровень выше верхнего допустимого";}
if(currentLevel > lowerLevel) && !(currentLevel > upperLevel)
   {message = " Уровень в норме";}
```

**Рис. 6.16.** Решение задачи о водонапорной башне на основе модели на рис. 6.15

### 6.2.3. Вложенные операторы `if` типа «два условия — три блока»

Возможны два варианта вложенных операторов `if` типа «два условия — три блока». На рис. 6.17 приведен первый вариант. Этот вариант образуется, если в блок `true`-ветви внешнего оператора `if` типа «одно условие — два блока» вложить внутренний оператор `if` типа «одно условие — два блока».




```
if(c1){
   if(c2)
      {<блок 1>}
   else
      {<блок 2>}
}
else
   {<блок 3>}
<последующие предложения>
```

**Рис. 6.17.** Вложенный оператор `if` типа «два условия — три блока», вариант 1. `c1` — условие ветвления внешнего оператора; `c2` — условие ветвления внутреннего оператора

Поскольку модель на рис. 6.17 содержит три блока, то необходимы три продукционных правила, регламентирующих их выбор. Анализ модели позволяет записать эти правила в следующем виде.

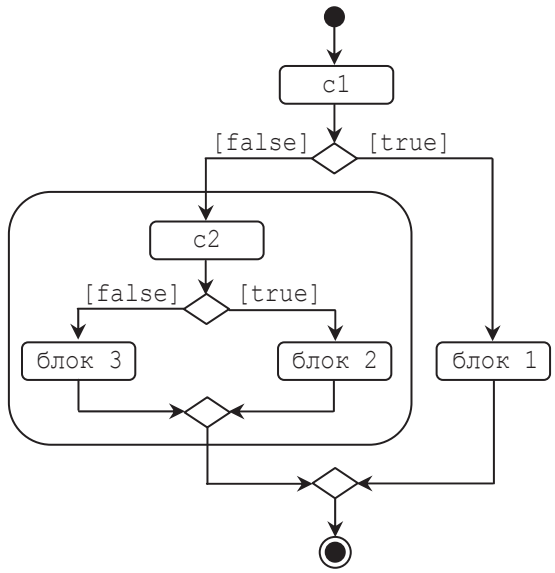
```
c1 && c2 ⇒ <блок1>
c1 && (!c2) ⇒ <блок2>
!c1 ⇒ <блок3>
```

Блок 1 выполняется только в том случае, когда условия ветвления и внешнего, и внутреннего операторов `if` принимают значение `true`. Блок 2 выполняется только в случае, когда условие ветвления внешнего оператора `if` принимает значение `true` и условие ветвления внутреннего оператора `if` — значение `false`. Блок 3 выполняется, когда условие ветвления внешнего оператора `if` принимает значение `false`.

На рис. 6.18 приведен второй вариант вложенного оператора `if` типа «два условия — три блока». Этот вариант образуется, если в блок `false`-ветви внешнего оператора `if` типа «одно условие — два блока» вложить внутренний оператор `if` такого же типа.

Продукционные правила, регламентирующие выполнение блоков для вложенного оператора `if` на рис. 6.18, имеют вид.

```
c1 ⇒ <блок1>
(!c1) && c2 ⇒ <блок2>
(!c1) && (!c2) ⇒ <блок3>
```




```
if(c1)
   {<блок 1>}
else {
   if(c2)
      {<блок 2>}
    else
      {<блок 3>}
}
<последующие предложения>
```

**Рис. 6.18.** Вложенный оператор `if` типа «два условия — три блока», вариант 2. `c1` — условие ветвления внешнего оператора; `c2` — условие ветвления внутреннего оператора

Блок 1 выполняется только в том случае, когда условие ветвления внешнего оператора `if` принимает значение `true`.

Блок 2 выполняется в том случае, когда условие ветвления внешнего оператора `if` принимает значение `false`, а условие ветвления внутреннего операторах `if` — значение `true`.

Блок 3 выполняется, если условия ветвления в обоих операторах `if` принимают значение `false`.

Оба варианта вложенного оператора `if` типа «два условия — три блока» могут использоваться для решения одной и той же *задачи императивного программирования, заключающейся в выборе одного из трех альтернативных блоков в том случае, когда выбор детерминируется двумя независимыми условиями*.

Проиллюстрируем справедливость этого утверждения на примере кодирования модифицированной задачи о водонапорной башне. Пусть, при сохранении прежнего условия задачи, диспетчер должен получать одно из трех альтернативных сообщений: `"Уровень выше верхнего допустимого"`, `"Уровень в норме"` и `"Уровень ниже нижнего допустимого"`. На рис. 6.19 приведены примеры кода, решающие сформулированную задачу на основе вариантов вложенной инструкции `if` типа «два условия — три блока», приведенных на рис. 6.17 и 6.18.

На рис. 6.19 предложения, при помощи которых объявляются и инициализируются необходимые переменные, не показаны. Приведены только вложенные операторы `if`, моделирующие рассуждения, на основе которых формируются сообщения.

```
if(currentLevel > lowerLevel)                    // внешний оператор if
   {if(currentLevel > upperLevel)                // внутренний оператор if
      {message = "Уровень выше верхнего допустимого";}
    else
      {message = "Уровень в норме";}}
else
   {message = "Уровень ниже нижнего допустимого";}
// последующие предложения
```

а)

```
if(currentLevel > upperLevel)                    // внешний оператор if
   {message = "Уровень выше верхнего допустимого";}
else
   {if(currentLevel > lowerLevel)                // внутренний оператор if
      {message = "Уровень в норме";}
    else
      {message = "Уровень ниже нижнего допустимого";}}
// последующие предложения
```

б)

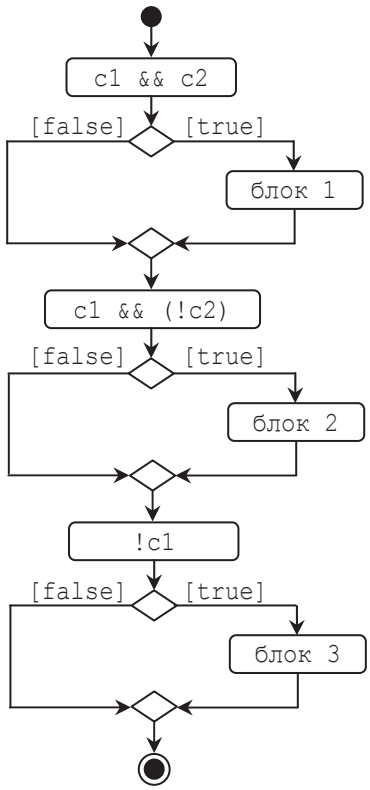
**Рис. 6.19.** Решение задачи о водонапорной башне с использованием вариантов вложенного оператора `if` типа «два условия — три блока».
а) соответствует варианту 1 на рис. 6.17; б) соответствует варианту 2 на рис. 6.18

На рис. 6.19 а) сообщения формируются на основе следующих рассуждений. Вначале проверяется, не является ли текущий уровень воды в резервуаре выше нижнего допустимого. Если это не так, то это означает, что уровень воды ниже нижнего допустимого и формируется сообщение `"Уровень ниже нижнего допустимого"`. Если это так, то осуществляется вторая проверка и проверяется, не является ли текущий уровень выше верхнего допустимого. Если из второй проверки следует, что текущий уровень выше верхнего допустимого, то формируется сообщение `"Уровень выше верхнего допустимого"`. Если из второй проверки следует, что текущий уровень ниже верхнего допустимого, то из этого следует, что он находится между двумя допустимыми уровнями и формируется сообщение `"Уровень в норме"`.

На рис. 6.19 б) вложенные операторы `if` моделируют следующие рассуждения. При помощи внешнего оператора `if` проверяется, не является ли текущий уровень воды в резервуаре выше верхнего допустимого. Если это так, то формируется сообщение `"Уровень выше верхнего допустимого"`. Если это не так, то при помощи внутреннего оператора `if` осуществляется вторая проверка и проверяется, не является ли текущий уровень выше нижнего допустимого. Если из второй проверки следует, что текущий уровень выше нижнего допустимого, то из этого следует, что он находится между верхним и нижним допустимыми уровнями и формируется сообщение `"Уровень в норме"`. Если из второй проверки следует,

что текущий уровень ниже нижнего допустимого, то формируется сообщение "Уровень ниже нижнего допустимого".

Задачу выбора одного из трех альтернативных блоков в тех случаях, когда условие выбора детерминируется двумя независимыми условиями, можно решать при помощи трех последовательно расположенных операторов if типа «одно условие — один блок». При этом условием ветвление каждого из операторов является левая часть соответствующего продукционного правила. На рис. 6.20 приведена модель и структура программного кода, логически эквивалентные вложенному оператору if на рис. 6.17.




```
if(c1 && c2)
   {<блок 1}
if(c1 && (!c2))
   {<блок 2}
if(!c1)
   {<блок 3}
```

**Рис. 6.20.** Три последовательно расположенных оператора if типа «одно условие — один блок» вместо вложенного оператора if типа «два условия — три блока»

Достоинства и недостатки модели, приведенной на рис. 6.20, точно такие же, как и у модели на рис. 6.15. Они иллюстрируются кодом, приведенным на рис. 6.21.

Проиллюстрируем работу модели на рис. 6.20 для решения модифицированной задачи о водонапорной башне, предполагающей формирование одного из трех альтернативных сообщений. Пусть, как и прежде

c1 соответствует currentLevel > lowerLevel
c2 соответствует currentLevel > upperLevel

тогда фрагмент программного кода, решающего модифицированную задачу о водонапорной башне, в соответствии с моделью на рис 6.20, можно представить рис. 6.21.

Ясно, что если уровень воды в резервуаре выше верхнего допустимого, то первый оператор `if` на рис. 6.21 сформирует требуемое сообщение, и нет необходимости в выполнении последующих операторов `if`. Однако структура кода такова, что в любом случае будут последовательно выполнены все три оператора `if`.

```
if((currentLevel > lowerLevel) && (currentLevel > upperLevel))
   {message = "Уровень выше верхнего допустимого";}
if((currentLevel > lowerLevel) && !(currentLevel > upperLevel))
   {message = "Уровень в норме";}
if(!(currentLevel > lowerLevel)
   {message = "Уровень ниже нижнего допустимого";}
```

**Рис. 6.21.** Решение модифицированной задачи о водонапорной башне на основе модели, приведенной на рис. 6.20

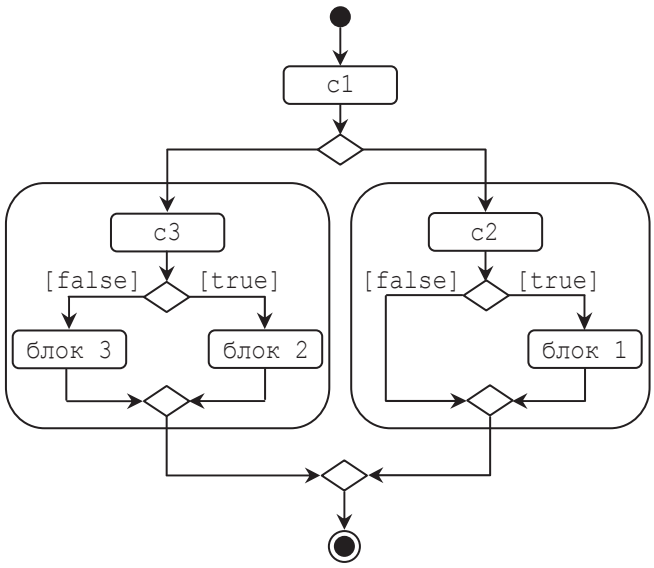
### 6.2.4. Вложенные операторы `if` типа «три условия — три блока»

Возможны два варианта вложенных операторов `if` типа «три условия — три блока». Эти варианты образуются, когда вложение осуществляется в оба блока внешнего оператора `if` типа «одно условие — два действия». На рис. 6.22 приведен первый вариант, полученный путем вложения в блок `true`-ветви внешнего оператора `if` оператора типа «одно условие — один блок», а в блок `false`-ветви внешнего оператора `if`-оператора типа «одно условие — два блока».

Продукционные правила для вложенного оператора `if` на рис. 6.22 запишем в виде.

```
c1 && c2 ⇒ <блок1>
(!c1) && c3 ⇒ <блок2>
(!c1) && (!c3) ⇒ <блок3>
```

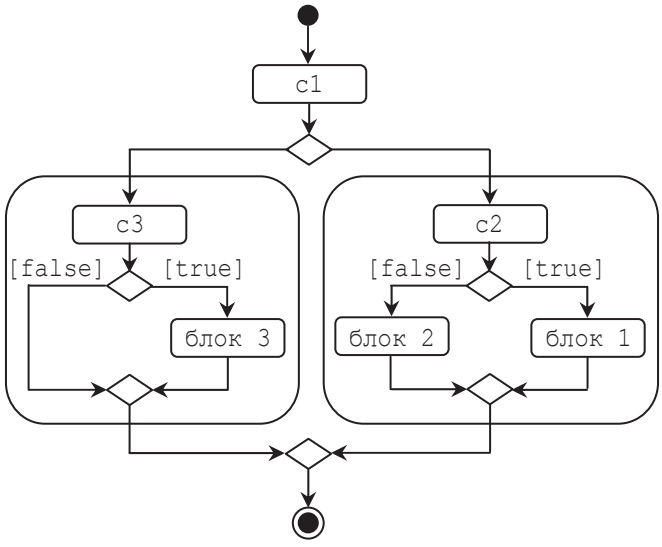
Блок 1 выполняется только в том случае, когда условие ветвления внешнего оператора `if` принимает значение `true` и условие ветвления правого внутреннего оператора `if` также принимает значение `true`. Блок 2 выполняется только в случае, когда условие ветвления внешнего оператора `if` принимает значение `false`, а условие ветвления левого внутреннего оператора `if` — значение `true`. Блок 3 выполняется при условии, что условие ветвления внешнего оператора `if` принимает значение `false` и условие ветвления левого внутреннего оператора `if` также принимает значение `false`.




```
if(c1){
   if(c2)
      {<блок 1>}
}
else{
   if(c3)
      {<блок 2>}
   else
      {<блок 3>}
}
<последующие предложения>
```

**Рис. 6.22.** Вложенный оператор `if` типа «три условия — три блока», вариант 1. `c1` — условие ветвления внешнего оператора; `c2` — условие ветвления правого внутреннего оператора; `c3` — условие ветвления левого внутреннего оператора

На рис. 6.23 приведен второй вариант вложенного оператора `if` типа «три условия — три блока». Этот вариант получен путем вложения в блок `true`-ветви




```
if(c1){
   if(c2)
      {<блок 1>}
   else
      {<блок 2>}
}
else{
   if(c3)
      {<блок 3>}
}
<последующие предложения>
```

**Рис. 6.23.** Вложенный оператор `if` типа «три условия — три блока», вариант 2. `c1` — условие ветвления внешнего оператора; `c2` — условие ветвления правого внутреннего оператора; `c3` — условие ветвления левого внутреннего оператора

внешнего оператора `if` оператора типа «одно условие — два блока», а в блок `false`-ветви внешнего оператора `if` — оператора типа «одно условие — один блок». Продукционные правила для вложенного оператора `if` на рис. 6.23 запишем в виде.

```
c1 && c2 ⇒ <блок1>
c1 && (!c2) ⇒ <блок2>
(!c1) && c3 ⇒ <блок3>
```

Блок 1 выполняется в том случае, когда и условие ветвления внешнего оператора `if`, и условие ветвления правого внутреннего оператора `if` принимают значение `true`.

Блок 2 выполняется в том случае, когда условие ветвления внешнего оператора `if` принимает значение `true`, а условие ветвления правого внутреннего оператора `if` принимает значение `false`.

Блок 3 выполняется в том случае, когда условие ветвления внешнего оператора `if` принимает значение `false`, а условие ветвления левого внутреннего оператора `if` принимает значение `true`.

Оба структурных варианта вложенного оператора `if` типа «три условия — три блока» предназначены для решения той же задачи выбора одного из трех альтернативных блоков, что и структурные варианты вложенного оператора `if` типа «два условия — три блока». Однако, наличие трех условий ветвления позволяет кодировать более изощренные рассуждения.

Проиллюстрируем применимость вложенного оператора `if` типа «три условия — три блока» на примере решения задачи о водонапорной башне, немного усложнив систему контроля. Будем считать, что система контроля использует показания двух датчиков: датчика работы насоса и датчика фактического уровня воды в резервуаре. В зависимости от показаний этих датчиков система контроля формирует и передает диспетчеру одно из трех сообщений.

Сообщение 1: `"Уровень выше верхнего допустимого. Насос включен. Аварийная ситуация."`. Это сообщение формируется в том случае, когда насос работает, а уровень воды в резервуаре превышает верхний допустимый уровень.

Сообщение 2: `"Уровень ниже нижнего допустимого. Насос выключен. Аварийная ситуация."`. Это сообщение формируется в том случае, когда насос не работает, а уровень воды в резервуаре опустился ниже нижнего допустимого уровня.

Сообщение 3: `"Уровень выше нижнего допустимого. Насос выключен"`. Это сообщение формируется в том случае, когда уровень воды в резервуаре выше нижнего допустимого уровня и насос выключен.

На рис. 6.24 приведены пример кода, решающие сформулированную задачу на основе первого варианта вложенного оператора `if` типа «три условия — три блока», приведенного на рис. 6.22.

```
boolean pump;                                    // предложение 1
float lowerLevel,
      upperLevel,
      currentLevel;                              // предложение 2
String message;                                  // предложение 3
// инициализация lowerLevel, upperLevel
// актуализация currentLevel и pump
if(pump){                                        // внешний if
   if(currentLevel > upperrLevel)                // правый внутренний if
      {message = "Уровень выше верхнего допустимого. Насос включен.
                  Аварийная ситуация.";}
}
else{
   if(currentLevel > lowerLevel)                 // левый внутренний if
      {message = "Уровень выше нижнего допустимого. Насос выключен.";}
   else
      {message = "Уровень ниже нижнего допустимого. Насос выключен.
                  Аварийная ситуация";}
}
// последующие предложения
```

**Рис. 6.24.** Решение задачи о водонапорной башне с использованием первого варианта вложенного оператора `if` типа «три условия — три блока»

Предложение 1 на рис. 6.24 объявляет переменную с именем `pump`, значение которой характеризует состояние насоса. Переменная `pump` принимает значение `true`, если насос включен и работает, и значение `false`, если насос выключен.

Предложение 2 объявляет три переменные типа `float`: для хранения значений допустимого нижнего уровня воды в резервуаре (переменная с именем `lowerLevel`), допустимого верхнего уровня воды в резервуаре (переменная с именем `upperLevel`) и текущего уровня воды в резервуаре (переменная с именем `currentLevel`).

Предложение 3 объявляет ссылочную переменную с именем `message` типа `String`. Эта переменная в дальнейшем будет ссылаться на строку, при помощи которой оператор информируется об уровне воды в резервуаре и состоянии насоса. Последующая часть кода моделирует рассуждения, в результате которых формируется одно из трех альтернативных сообщений.

Опишем условия ветвления операторов `if` на рис. 6.22 следующими булевыми выражениями.

`c1` соответствует `pump`
`c2` соответствует `currentLevel > upperLevel`
`c3` соответствует `currentLevel > lowerLevel`

На рис. 6.24 сообщения формируются на основе следующих рассуждений. Вначале проверяется, включен ли насос. Если насос включен, то осуществляется проверка текущего уровня воды в резервуаре. Если текущий уровень воды выше верхнего допустимого уровня, то формируется сообщение: "Уровень выше верхнего допустимого. Насос включен. Аварийная ситуация". Если насос выключен, то опять осуществляется проверка текущего уровня воды в резервуаре. Если текущий уровень воды в резервуаре выше нижнего допустимого уровня, то формируется сообщение: "Уровень выше нижнего допустимого. Насос выключен.". Если при выключенном насосе текущий уровень воды ниже нижнего допустимого уровня, то формируется сообщение: "Уровень ниже нижнего допустимого. Насос выключен. Аварийная ситуация."

Как и во всех ранее рассмотренных случаях, вместо одного вложенного оператора `if` типа «три условия — три блока» можно использовать три, последовательно расположенных оператора `if` типа «одно условие — один блок». Код, приведенный на рис. 6.25, также является решением задачи о водонапорной башне для случая двух датчиков и выполняет ту же работу, что код, приведенный на рис. 6.24.

```
if((pump)&&(currentLevel > upperrLevel))
   {message = "Уровень выше верхнего допустимого. Насос включен.
               Аварийная ситуация.";}
if(!pump)&&(currentLevel > lowerLevel))
   {message = "Уровень выше нижнего допустимого. Насос выключен.";}
if(!pump)&&(!(currentLevel > lowerLevel))
   {message = "Уровень ниже нижнего допустимого. Насос выключен.
               Аварийная ситуация.";}
```

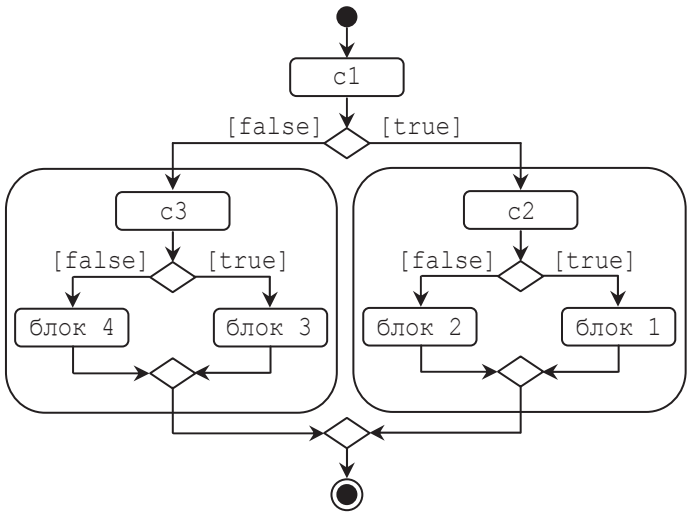
**Рис. 6.25.** Решение задачи о водонапорной башне для случая двух датчиков при помощи трех последовательных операторов `if` типа «одно условие — один блок»

### 6.2.5. Вложенный оператор `if` типа «три условия — четыре блока»

Возможен только один вариант вложенного оператора `if` типа «три условия — четыре блока», который образуется при вложении в оба блока внешнего оператора `if` типа «одно условие — два блока» внутренних операторов `if` такого же типа. На рис. 6.26 приведены модель и структура программного кода вложенного оператора `if` типа «три условие — четыре блока».

Поскольку модель на рис. 6.26 включает четыре блока, то выбор и выполнение блоков регламентируются четырьмя продукционными правилами.

```
c1 && c2 ⇒ <блок1>
c1 && (!c2) ⇒ <блок2>
(!c1) && c3 ⇒ <блок3>
(!c1) && (!c3) ⇒ <блок4>
```




```
if(c1){
   if(c2)
      {<блок 1>}
   else
      {<блок 2>}
}
else{
   if(c3)
      {<блок 3>}
   else
      {<блок 4>}
}
<последующие предложения>
```

**Рис. 6.26.** Вложенный оператор `if` типа «три условия — четыре блока». `c1` — условие ветвления внешнего оператора; `c2` — условие ветвления правого внутреннего оператора; `c3` — условие ветвления левого внутреннего оператора

Вложенный оператор `if` типа «три условия — четыре блока» может использоваться *для решения задачи императивного программирования, заключающейся в выборе одного из трех альтернативных блоков в том случае, когда выбор детерминируется тремя независимыми условиями*.

Проиллюстрируем работу вложенного оператора `if` типа «три условия — четыре блока» на примере решения задачи о водонапорной башне для случая двух датчиков, рассмотренную ранее (см. рис. 6.24). Усложним формулировку задачи тем, что увеличим количество сообщений, которые получает диспетчер. Пусть теперь система контроля передает диспетчеру одно из следующих четырех сообщений:

```
"Уровень выше верхнего допустимого. Насос включен. Аварийная ситуация.";
"Уровень ниже верхнего допустимого. Насос включен.";
"Уровень ниже нижнего допустимого. Насос выключен. Аварийная ситуация";
"Уровень выше нижнего допустимого. Насос выключен.".
```

Опишем условия ветвления операторов `if` на рис. 6.26 следующими булевыми выражениями.

`c1` соответствует `pump`
`c2` соответствует `currentLevel > upperLevel`
`c3` соответствует `currentLevel > lowerLevel`

Пример кода задачи о водонапорной башне для случая двух датчиков и четырех сообщений приведен на рис. 6.27.

```
boolean pump;                                                    // предложение 1
float lowerLevel,
      upperLevel,
      currentLevel;                                              // предложение 2
String message;                                                  // предложение 3
// инициализация lowerLevel, upperLevel
// актуализация currentLevel и pump
if(pump){                                                        // внешний if
   if(currentLevel > upperrLevel)                                // правый if
      {message = "Уровень выше верхнего допустимого. Насос включен.
                  Аварийная ситуация.";}
   else
      {message = "Уровень ниже верхнего допустимого. Насос включен.";}
}
else{
   if(currentLevel > lowerLevel)                                 // левый if
      {message = "Уровень выше нижнего допустимого. Насос выключен.";}
   else
      {message = "Уровень ниже нижнего допустимого. Насос выключен.
                  Аварийная ситуация";}
}
// последующие предложения
```

**Рис. 6.27.** Кодирование задачи о водонапорной башне с использованием вложенного оператора `if` типа «три условия — четыре блока»

Ясно, что для выбора одного из четырех альтернативных блоков не обязательно использовать вложенный оператор `if` типа «три условия — четыре блока». Такой выбор можно осуществить при помощи четырех, следующих друг за другом операторов `if` типа «одно условие — один блок».

## 6.3. Каскадное вложение оператора `if`

При помощи вложенных операторов `if` решаются задачи императивного программирования, заключающиеся в выборе и выполнении одного из нескольких альтернативных блоков. Ограничившись степенью вложения, равной двум, мы получили девять различных моделей и объединили их в пять типов. При этом наибольшее количество блоков, подлежащих выбору, не превышало четыре, для вложенного оператора `if` типа «три условия — четыре блока».

Ясно, что в практике императивного программирования может возникнуть *задача выбора одного из многих альтернативных блоков*, количество которых превышает четыре. Эту задачу можно решать несколькими способами.

Во-первых, можно развивать идею вложенности и рассматривая варианты, порождаемые в тех случаях, когда степень вложенности превышает два. Такой подход возможен, однако, он не является наилучшим. Недостатком этого подхода является то, что «логика выбора» блока определяется индивидуальной и трудно воспринимаемой структурой вложения операторов `if`. При отображении моделей вложенных операторов `if` в программный код, в тех случаях, когда степень вложения превышает два, код, во-первых, трудно воспринимаем, а во-вторых, при кодировании легко совершить ошибку.

Другим способом решения задачи выбора одного из многих альтернативных блоков является использование последовательности независимых операторов `if` типа «одно условие — один блок». При использовании этого способа «логика выбора» определяется булевым выражением каждого из операторов `if`. Этот способ обладает тем достоинством, что отображается в логически простую и регулярную структуру. Недостаток этого способа известен, и он заключается в том, что все операторы `if`, составляющие последовательность, должны обязательно выполняться.

Существует еще один способ решения задачи выбора одного из множества альтернативных блоков. Этот способ, с одной стороны, развивает идею вложенности, а с другой — использует регулярную и легко воспринимаемую структуру кода. Этот способ часто называют *каскадное вложение оператора* `if`. Кроме регулярности и легкости восприятия, каскадное вложение оператора `if` обладает еще и тем достоинством, что если происходит выбор одного из блоков каскада, то процедура выбора завершается, и все оставшиеся операторы `if` не выполняются. Каскадное вложение оператора `if` можно рассматривать как развитие модели, приведенной на рис. 6.18 (вложенный оператор `if` типа «два условия — три блока», вариант 2). Каскадное вложение оператора `if` образуется, если в блок `false`-ветви исходного оператора `if` типа «одно условие — два блока» многократно вкладывать оператор `if` такого же типа. На рис. 6.28 приведена модель каскадного вложения оператора `if`.

На рис. 6.28 степень вложенности оператора `if` равна четырем и обеспечивает выбор одного из пяти альтернативных блоков. Ясно, что если необходимо осуществить выбор из большего количества блоков, то необходимо увеличить степень вложенности, сохраняя его принцип.

На рис. 6.29 приведена структура кода, соответствующая модели на рис. 6.28. Синтаксической особенностью программного кода, приведенного на рис. 6.29 является то, что фигурными скобками выделяются только выбираемые блоки. Блоки, формируемые в результате вложения и изображенные на рис. 6.28, не заключаются в фигурные скобки.

В том случае, когда выбирается блок, который состоит из одного предложения, он может не заключатся в фигурные скобки.

При выполнении кода, приведенного на рис. 6.29, булевы выражения, соответствующие условиям ветвления, проверяются сверху вниз, начиная с первого.

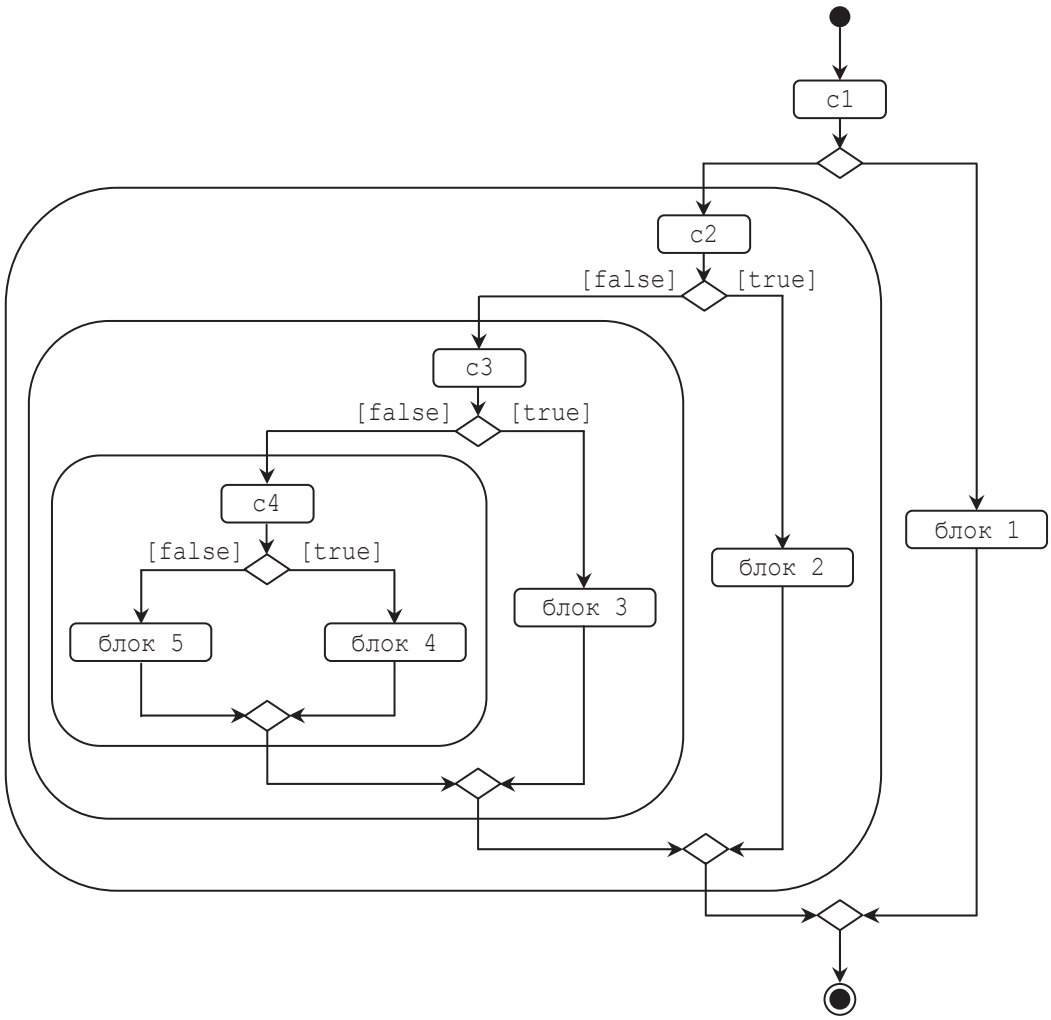



**Рис. 6.28.** Модель каскадного вложения оператора `if`. `c1`–`c4` условия ветвления

```
if(c1)
   {<блок 1>}
else if(c2)
   {<блок 2>}
else if(c3)
   {<блок 3>}
else if(c4)
   {<блок 4>}
else
   {<блок 5>}
<последующие предложения>
```

**Рис. 6.29.** Структура кода при каскадном вложении оператора `if`

Как только встречается булево выражение со значением `true`, то соответствующий блок выполняется, все остальные операторы `if` игнорируются, и начинают выполняться последующие предложения. Блок 5 выполняется в том случае, когда все булевы выражения приняли значение `false`.

Проиллюстрируем каскадное вложение оператора `if` на примере. В качестве задачи, подлежащей кодированию, рассмотрим задачу формирования буквенной оценки уровня знаний студентов, которая принята в ряде стран. Знания оцениваются в два этапа. Вначале определяется общее количество баллов, которые студент получил, выполняя различные учебные задания в семестре и в ходе семестрового экзамена. Это количество баллов может колебаться в диапазоне от 0 до 100. Затем, в зависимости от общего количества баллов, уровень знаний оцениваются при помощи одного или двух буквенных символов. Например, если общее количество баллов находится в диапазоне от 90 до 100, то уровень знаний оцениваются буквой «A». Это наивысшая оценка. Общий список буквенных символов, используемых для оценки уровня знаний, состоит из следующих символов, расположенных в порядке убывания оценки: «A», «B», «C», «D», «E», «FX», «F».

На рис. 6.30 приведен программный код, который решает сформулированную задачу, используя каскадное вложение оператора `if`.

```
int score;                          // предложение 1
String grade;                       // предложение 2
// инициализация score
   if(score >= 90)                  // 1-й оператор if
      {grade = "A";}                // блок 1
   else if(score >= 85)             // 2-й оператор if
      {grade = "B";}                // блок 2
   else if(score >= 75)             // 3-й оператор if
      {grade = "C";}                // блок 3
   else if(score >= 68)             // 4-й оператор if
      {grade = "D";}                // блок 4
   else if(score >= 60)             // 5-й оператор if
      {grade = "E";}                // блок 5
   else if(score >= 35)             // 6-й оператор if
      {grade = "FX";}               // блок 6
   else
      {grade = "F";}                // блок 7
```

**Рис. 6.30.** Решение задачи оценки знаний при помощи каскадного вложения оператора `if`

Предложение 1 объявляет целочисленную переменную с именем `score`. Эта переменная предназначена для хранения общего количества баллов в диапазоне 0–100. Предложение 2 объявляет ссылочную переменную с именем `grade` типа `String`. Эта переменная в дальнейшем будет использоваться для ссылки на строку, содержащую

буквенную оценку знаний. После инициализации переменной score следует конструкция каскадного вложения оператора if. Булевы выражения сформированы с учетом последовательного (сверху вниз) выполнения оператора if. Так, например, оценку «B» студент получает в том случае, когда его общее количество баллов находится в диапазоне от 85 до 90. Однако во втором операторе if булево выражение имеет вид (score >= 85), а не ((score >= 85)&&(score < 90)), поскольку если выполнился второй оператор if, то это означает, что булево выражение в первом операторе if равно false. Каждый из блоков представляет собой предложение, создающее строку с именем grade. Очевидно, что блок 7 выполняется тогда, когда значение переменной score находится в диапазоне от 0 до 34.

## 6.4. Оператор switch

Каскадное вложение оператора if является универсальным средством выбора одного блока из множества альтернативных блоков. Условие выбора любого из блоков может осуществляться *индивидуальной «логикой выбора»*, которая детерминируется соответствующим булевым выражением. Универсальность каскада операторов if определяется тем, что все булевы выражения могут быть различны.

В ряде случаев «логика выбора» может быть одинаковой для всех блоков. Это может происходить, например, когда выбор блока детерминируется различными значениями одного и того же выражения. На рис. 6.31 приведена структура каскадного вложения оператора if, иллюстрирующая случай выбора блоков на основании различных значений одного и того же выражения.

```
if(<выражение> == <литерал 1>)
   {<блок 1>}
else if(<выражение> == <литерал 2>)
   {<блок 2>}
else if(<выражение> == <литерал 3>)
   {<блок 3>}
else if(<выражение> == <литерал 4>)
   {<блок 4>}
else
   {<блок 5>}
<последующие предложения>
```

**Рис. 6.31.** Структура кода для каскадного вложения оператора if с одинаковой «логикой выбора» блоков

Код, приведенный на рис. 6.31, обеспечивает выбор одного из блоков 1–4, в зависимости от возвращаемого значения одного и того же выражения. Каждое из

булевых выражений сравнивает возвращаемое значение этого выражения с литералом. Булево выражение принимает значение `true` в том случае, если возвращаемое значение выражения равно значению соответствующего литерала. Блок 5 выполняется в том случае, когда все булевы выражения приняли значение `false`. Это означает, что возвращаемое значение выражения не совпало ни с одним из литералов.

В тех случаях, когда выбор одного из множества блоков осуществляется на основании различных значений одного и того же выражения, вместо каскада операторов `if` можно использовать оператор `switch`, структура которой приведена на рис. 6.32.

```
switch(<выражение>){
case <литерал 1>:
   {<блок 1>}
   break;
case <литерал 2>:
   {<блок 2>}
   break;
case <литерал 3>:
   {<блок 3>}
   break;
case <литерал 4>:
   {<блок 4>}
   break;
default:
   {<блок по умолчанию>}
}
<последующие предложения>
```

**Рис. 6.32.** Структура оператора `switch`

На рис. 6.32 в первой строке, после служебного слова `switch`, в круглых скобках, задается выражение, возвращаемое значение которого определяет выбор одного из множества блоков. Возвращаемое значение этого выражения должно иметь один из следующих типов: `byte`, `short`, `int`, `char` или `String`. Затем в фигурных скобках располагается набор `case`-структур, осуществляющих выбор. Количество `case`-структур равно количеству выбираемых блоков. Оператор `switch` на рис. 6.32 позволяет осуществить выбор одного из четырех блоков.

Каждая `case`-структура начинается со служебного слова `case`, за которым следует значение выражения, детерминирующее выбор блока. Это значение задается литералом, тип которого соответствует типу возвращаемого значения выражения либо типу, совместимому с типом возвращаемого значения выражения. Во всех `case`-структурах должны быть записаны различные литералы. Дублирование литералов недопустимо. После литерала следует двоеточие, за которым размещается

выбираемый блок. Блок выбирается в том случае, если возвращаемое значение выражения равно литералу. Завершается case-структура необязательным оператором прерывания break. Оператор break будет изучаться более подробно в разделе, посвященном программированию итераций.

После последней case-структуры располагается необязательная default-структура, содержащая блок, выбираемый по умолчанию. Если оператор switch содержит default-структуру, то блок, выбираемый по умолчанию, выполняется в том случае, если ни в одной из предшествующих case-структур не произошел выбор блока.

Выбор блока при помощи case-структур происходит последовательно сверху вниз. Если для данной case-структуры возвращаемое значение выражения не совпадает с литералом, то осуществляется переход к следующей case-структуре. Если в какой-либо из case-структур происходит совпадение возвращаемого значения выражения с литералом, то выбирается и выполняется блок, включенный в эту case-структуру. После завершения выполнения предложений блока при помощи оператора break осуществляется прерывание дальнейшего выполнения оператора switch, оставшиеся case-структуры и default-структура игнорируются и выполняются предложения, следующие после оператора switch.

Таким образом, при решении задачи императивного программирования, заключающейся в выборе одного блока из множества альтернативных блоков, в тех случаях, когда критерием выбора является возвращаемое значение некоторого выражения, можно использовать как каскадное вложение оператора if, так и оператор switch. Предпочтение в этом случае следует отдавать оператору switch, поскольку выполнение байт кода, полученного в результате трансляции оператора switch, осуществляется быстрее, чем выполнение байт кода, полученного в результате трансляции каскада операторов if.

Проиллюстрируем использование оператора switch на примере решения следующей задачи. Пусть некоторая целочисленная переменная принимает значения в диапазоне от 1 до 7, которые интерпретируются как номера дней недели. В зависимости от конкретного значения этой переменной необходимо создать строку с наименованием дня недели. Если, например, значение переменной равно 1, то необходимо создать строку "понедельник". Если целочисленная переменная принимает значения, находящиеся вне диапазона 1–7, то должна быть сформирована строка со значением "неизвестный день". Код на рис. 6.33 является возможным решением сформулированной задачи.

На рис. 6.33 предложение 1 объявляет целочисленную переменную с именем dayNumber, которая принимает значения, соответствующие номеру дня недели. Предложение 2 объявляет ссылочную переменную типа String с именем dayName, которая в дальнейшем будет использоваться для создания строки с наименованием дня недели.

Затем, после инициализации переменной dayNumber, располагается оператор switch, который включает семь case-структур и default-структуру. В случае совпадения значения переменной dayNumber с литералом, указанным в case-структуре,

выбирается и выполняется соответствующий блок. Блок состоит из одного предложения, которое создает строку dayName с предопределенным наименованием дня недели. Если значение переменной dayNumber таково, что ни одна case-структура не осуществляет выбор блока, то работает default-структура, которая создает строку dayName со значением "неизвестный день".

```
byte dayNumber;                          // предложение 1
String dayName;                          // предложение 2
// инициализация dayNumber
switch(dayNumber){
case 1: dayName = "понедельник";
        break;
case 2: dayName = "вторник";
        break;
case 3: dayName = "среда";
        break;
case 4: dayName = "четверг";
        break;
case 5: dayName = "пятница";
        break;
case 6: dayName = "суббота";
        break;
case 7: dayName = "воскресенье";
        break;
default: dayName = "неизвестный день";
}
```

**Рис. 6.33.** Решение задачи о наименованиях дней недели на основе оператора switch

Использование оператора break в case-структурах не является обязательным. Если он отсутствует, то после проверки совпадения возвращаемого значения с литералом, заданным текущей case-строкой, работа оператора switch не прерывается, и осуществляется переход к следующей case-строке. В этом случае оператор switch осуществляет выбор блока, если возвращаемое значение выражения равно *одному из нескольких литералов*. Иными словами, выбор блока осуществляется, если возвращаемое значение выражения принадлежит подмножеству возможных значений.

Проиллюстрируем работу оператора switch, в котором опущено некоторое количество операторов break, на примере решения следующей задачи. Пусть целочисленная переменная принимает значения от 1 до 12, которые интерпретируются как номера месяцев.

Если значение переменной равно или 1, или 2, или 12, то необходимо сформировать строку со значением "зима".

Если значение переменной равно или 3, или 4, или 5, то формируется строка со значением "весна".

Если переменная принимает одно из значений 6, 7 или 8, то формируется строка со значением "лето".

Если переменная принимает значение, равное или 9, или 10, или 11, то формируется строка со значением "осень".

Если значение переменной отлично от чисел 1–12, то формируется строка со значением "неизвестное время года".

На рис. 6.34 приведен код, иллюстрирующий решение сформулированной задачи. На рис. 6.34 предложение 1 объявляет целочисленную переменную с именем monthNumber, которая принимает значения, соответствующие номеру месяца. Предложение 2 объявляет ссылочную переменную seasonName, которая в дальнейшем будет использоваться для создания строки с наименованием времени года.

```
byte monthNumber;                          // предложение 1
String seasonName;                         // предложение 2
// инициализация monthNumber
switch(monthNumber){
case 1:
case 2:
case 12:
   {seasonName = "зима";}
   break;
case 3:
case 4:
case 5:
   {seasonName = "весна";}
   break;
case 6:
case 7:
case 8:
   {seasonName = "лето";}
   break;
case 9:
case 10:
case 11:
   {seasonName = "осень";}
   break;
default: {seasonName = "неизвестное время года";}
}
```

**Рис. 6.34.** Решение задачи о наименованиях времен года на основе оператора switch с пропущенными словами break

После инициализации переменной monthNumber располагается оператор switch, предназначенный для выбора одного из четырех блоков. Каждый из этих четырех блоков состоит из предложения, создающего строку с предопределенным значением. Рассмотрим, каким образом выбирается предложение, создающее строку "зима". Этому предложению предшествуют три служебных слова case, детерминирующие три целочисленных литерала: 1, 2 и 12. Такая конструкция должна интерпретироваться следующим образом. Предложение, создающее строку "зима", выполняется в том случае, если возвращаемое значение выражения равно или 1, или 2, или 12. За этим предложением следует оператор break, который осуществляет прерывание работы оператора switch.

Аналогичным образом работают фрагменты кода, при помощи которых выбираются предложения, формирующие строки "весна", "лето" и "осень". В том случае, если возвращаемое значение выражения не принадлежит диапазону 1–12, выполняется default-структура, которая создает строку "неизвестное время года".

## 6.5. Вложенные операторы switch

Программный блок, выбираемый при помощи оператора switch, в свою очередь, может включать оператор switch. Такие конструкции называются *вложенными операторами* switch. На рис. 6.35 приведена структура кода, иллюстрирующая вложенный оператор switch.

```
                switch(<внешнее выражение>){
                case <литерал 1.1>:
                   switch(<внутреннее выражение>){
                   case <литерал 2.1>:
                      {<внутренний блок 1>}
                      break;
внешний блок 1     case <литерал 2.2>:
                      {<внутренний блок 2>}
                      break;
                   default:
                      {<внутренний блок по умолчанию>}
                   }
                   break;
                case <литерал 1.2>:
                   {<внешний блок 2>}
                   break;
                default:
                   {<внешний блок по умолчанию>}
                }
                <последующие предложения>
```

**Рис. 6.35.** Структура вложенного оператора switch

Структура программного кода, приведенная на рис. 6.35, соответствует случаю, когда и внешний, и внутренний операторы `switch` состоят из двух `case`-структур и включают `default`-структуру. Во внешний блок 1 вложен внутренний оператор `switch`.

Поскольку сам оператор `switch` представляет собой программный блок, заключенный в фигурные скобки, то не нужно выделять внутренний оператор `switch` при помощи фигурных скобок. Из того обстоятельства, что оператор `switch` представляет собой программный блок, следует, что во внешнем и внутреннем операторах `switch` допустимы одинаковые литералы. Прерывание выполнения внутреннего оператора `switch` при достижении оператора `break` приводит к выходу из внутреннего оператора `switch` во внешний оператор `switch` и выполнению оставшихся `case`-структур либо `default`-структуры внешнего оператора `switch`.

Применимость вложенного оператора `switch` определяется присущей ему «логикой выбора» внутреннего блока. На рис. 6.35 внутренний блок 1 выбирается в том случае, когда внешнее выражение принимает значение литерала 1.1 и внутреннее выражение принимает значение 2.1, а внутренний блок 2 выбирается в том случае, когда внешнее выражение принимает значение литерала 1.1 и внутреннее выражение принимает значение 2.2.

Проиллюстрируем использование вложенного оператора `switch` на примере решения следующей задачи. Пусть некоторое приложение последовательно выводит пользователю два вопроса, на которые пользователь должен ответить «да» или «нет». В зависимости от комбинации ответов на эти вопросы, система формирует одно из четырех альтернативных сообщений.

На рис. 6.36 приведен код, являющийся одним из возможных решений сформулированной задачи на основе вложенного оператора `switch`.

Предложение 1 на рис. 6.36 объявляет две переменные с именами `firstAnswer` и `secondAnswer`. Переменная `firstAnswer` хранит ответ на первый вопрос, а переменная `secondAnswer` — ответ на второй вопрос. Если ответ утвердительный, то значение любой из переменных равно литералу `'Y'`, а если ответ отрицательный, то значение любой из переменных равно `'N'`.

Предложение 2 объявляет переменную `message` типа `String`. Строка с этим именем должна быть сформирована в результате работы кода.

В качестве выражения во внешнем операторе `switch` фигурирует переменная `firstAnswer`. Оператор включает две `case`-структуры и `default`-структуру. Первая `case`-структура работает в том случае, когда переменная `firstAnswer` принимает значение `'Y'`, а вторая — в том случае, когда переменная `firstAnswer` принимает значение `'N'`. Если переменная `firstAnswer` принимает какое-либо иное значение, то работает `default`-структура внешней инструкции `switch` и формируется строка с именем `message` и значением «`не понятен первый ответ`».

В программный блок каждой из `case`-структур внешнего оператора `switch` вложен внутренний оператор `switch`. В качестве выражения в обоих внутренних операторах `switch` фигурирует переменная `secondAnswer`.

Каждый из внутренних операторов `switch` состоит из двух `case`-структур и `default`-структуры. Эти `case`-структуры работают так же, как и `case`-структуры внешнего оператора `switch`.

```
char firstAnswer,
     secondAnswer;            // предложение 1
String message;               // предложение 2
// инициализация firstAnswer и secondAnswer
switch(firstAnswer){          // внешний оператор switch
case 'Y':
   switch(secondAnswer){      // первый внутренний оператор switch
   case 'Y':
      {message = . . . }      // оба ответа «да»
      break;
   case 'N':
      {message = . . . }      // первый ответ «да», а второй ответ «нет»
      break;
   default:
      {message = «не понятен второй ответ»;}
   }
   break;
case 'N':
   switch(secondAnswer){      // второй внутренний оператор switch
   case 'Y':
      {message = . . . }      // первый ответ «нет», а второй ответ «да»
      break;
   case 'N':
      {message = . . . }      // оба ответа «нет»
      break;
   default:
      {message = «не понятен второй ответ»;}
   }
default:
   {message = «не понятен первый ответ»;}
}
```

**Рис. 6.36.** Решение задачи с двумя вопросами на основе вложенного оператора `switch`

Первая работает в том случае, когда переменная `secondAnswer` принимает значение `'Y'`, а вторая — в том случае, когда эта переменная принимает значение `'N'`. Если переменная `secondAnswer` принимает значение, отличное от `'Y'` или `'N'`, то работает `default`-структура внутреннего оператора `switch` и формируется строка `«не понятен второй ответ»`.

Оператор `break` может быть опущен как во внешнем, так и во внутреннем операторах `switch`. Ясно, что пропуск оператора `break` меняет логику выбора блоков.

## Упражнения для самостоятельной работы

6.1. Разработайте программный код, решающий следующую задачу. Даны три целых числа $a$, $b$, $c$. Если выполняется условие $a \le b \le c$, то необходимо все числа заменить их квадратами. Если это условие не выполняется, то необходимо изменить знак каждого числа на противоположный.

6.2. Разработайте программный код, который из трех числовых переменных $a$, $b$ и $c$ определит переменную, содержащую наибольшее значение.

6.3. Разработайте программный код, определяющий результат гадания на ромашке — "любит — не любит", взяв за исходное данное количество лепестков.

6.4. Разработайте программный код для решения следующей задачи. Известно, что из четырех числовых переменных `a1`, `a2`, `a3` и `a4` значение трех равны между собой. Определите переменную, значение которой отличается от значений остальных переменных. Введите переменную n и присвойте ей номер найденной переменной.

6.5. Разработайте программный код, который позволяет определить номер квадранта декартовой системы координат для точки с координатами $x$, $y$.

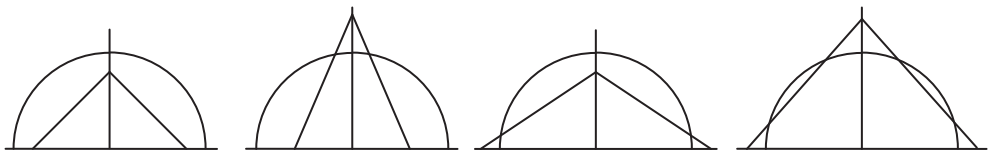
6.6. Разработайте программный код, который по координатам трех вершин некоторого прямоугольника (`x1`, `y1`), (`x2`, `y2`), (`x3`, `y3`), стороны которого параллельны осям координат OX и OY, позволит определить координаты четвертой вершины.

6.7. Разработайте программный код для определения типа треугольника с произвольно заданными длинами сторон $a$, $b$ и $c$ (произвольный, равнобедренный, равносторонний, не является треугольником).

6.8. Разработайте программный код, который для целого числа $k$, изменяющегося в пределах от 1 до 99, сформирует строку, в которой число $k$ интерпретируется как возраст. Например, «мне 4 года», «мне 10 лет», «мне 21 год». Код должен учитывать, что, в зависимости от значения числа $k$, нужно использовать одно из слов: «год», «года», «лет». При разработке кода используйте оператор `switch`.

6.9. Дан равнобедренный треугольник с основанием `a` и высотой `h` и полуокружность с радиусом $R$. Разработайте программный код, который позволяет определить: (1) находится ли треугольник полностью внутри полуокружности (рис. а); (2) выходит ли треугольник за пределы полуокружности

только своей вершиной (рис. б); (3) выходит ли треугольник за пределы окружности только своим основание (рис. в), (4) выходит ли треугольник за пределы окружности и вершиной, и основанием (рис. г). Если треугольник полностью находится внутри полуокружности, то определите, какой процент площади полуокружности он занимает.



а) б) в) г)

6.10. На основе оператора `switch` разработайте программный код, который по заданным единице измерения (килограмм, миллиграмм, грамм, тонна, центнер) и массе M рассчитывал бы соответствующее значение массы в килограммах.

6.11. На основе оператора `switch` разработайте программный код, который по последней цифре числа определит последнюю цифру его квадрата.

6.12. На основе оператора `switch` разработайте программный код простого калькулятора, выполняющего операции сложения, вычитания, умножения и деления. Для ввода операндов и вывода результата используйте консоль.

6.13. Имеется серия измерений элементов треугольника. Группы элементов пронумерованы. В серии в произвольном порядке могут встречаться такие группы элементов треугольника: (1) основание и высота; (2) две стороны и угол между ними (угол задан в радианах); (3) три стороны. Разработайте программу, которая запрашивает номер группы элементов, выводит их значения на экран и вычисляет площадь треугольника. При разработке кода используйте оператор `switch`.

6.14. Разработайте программный код, который по номеру некоторого года `year` (`year` > 0) определит номер столетия. При этом следует учесть, что столетие начинается в первом году, а не в нулевом.

Раздел 7

# ПРОГРАММИРОВАНИЕ ИТЕРАЦИЙ

В практике императивного программирования часто возникает необходимость многократно выполнять блок предложений, модифицируя при каждом повторении значение одной или нескольких переменных. Однократное выполнение блока предложений называется *итерация*, а процесс многократного повторения итераций — *итерационный процесс*. Для осуществления итерационного процесса в программном коде организуется *цикл*. Таким образом, целью организации цикла в программном коде является реализация итерационного процесса.

## 7.1. Компоненты цикла. Варианты организации цикла

Рассмотрим пример, который, с одной стороны, иллюстрирует необходимость в организации цикла, а с другой — позволит сформулировать основные компоненты цикла и действия, необходимые для реализации любого цикла. Задача состоит в вычислении площадей ряда прямоугольных треугольников. Катеты первого треугольника равны 3 и 4 сантиметрам, катеты каждого последующего треугольника больше, чем катеты предыдущего треугольника на 10%.

На рис. 7.1 приведен фрагмент кода, осуществляющий вычисление площадей прямоугольных треугольников для ряда, состоящего из пяти треугольников.

В коде, приведенном на рис. 7.1, можно выделить несколько типовых структурных элементов и действий над ними, которые являются общими для любой подобной программы.

Первое предложение кода объявляет две переменные с именами `cathetus1` (катет1) и `cathetus2` (катет2), которым присваиваются начальные значения. Это те переменные, которые в дальнейшем будут использоваться и модифицироваться при каждой итерации. Назовем эти переменные *параметрами цикла*. Присваивание параметрам исходных значений будем называть *начальной инициализацией параметров* или просто инициализацией параметров. Второе предложение объявляет и инициализирует переменную `triangleArea` (площадь треугольника), в которую последовательно будут записываться площади треугольников.

```
double cathetus1 = 3.0,       // объявление и инициализация 1-й переменной
       cathetus2 = 4.0;       // объявление и инициализация 2-й переменной
double triangleArea = 0.0;

triangleArea = (cathetus1 * cathetus2)/2.0;  // площадь 1-го треугольника
// запись значения triangleArea в таблицу
cathetus1 = cathetus1 + cathetus1 * 0.1;     // модификация 1-й переменной
cathetus2 = cathetus2 + cathetus2 * 0.1;     // модификация 2-й переменной

triangleArea = (cathetus1 * cathetus2)/2.0;  // площадь 2-го треугольника
// запись значения triangleArea в таблицу
cathetus1 = cathetus1 + cathetus1 * 0.1;     // модификация 1-й переменной
cathetus2 = cathetus2 + cathetus2 * 0.1;     // модификация 2-й переменной

triangleArea = (cathetus1 * cathetus2)/2.0;  // площадь 3-го треугольника
// запись значения triangleArea в таблицу
cathetus1 = cathetus1 + cathetus1 * 0.1;     // модификация 1-й переменной
cathetus2 = cathetus2 + cathetus2 * 0.1;     // модификация 2-й переменной

triangleArea = (cathetus1 * cathetus2)/2.0;  // площадь 4-го треугольника
// запись значения triangleArea в таблицу
cathetus1 = cathetus1 + cathetus1 * 0.1;     // модификация 1-й переменной
cathetus2 = cathetus2 + cathetus2 * 0.1;     // модификация 2-й переменной

triangleArea = (cathetus1 * cathetus2)/2.0;  // площадь 5-го треугольника
// запись значения triangleArea в таблицу
cathetus1 = cathetus1 + cathetus1 * 0.1;     // модификация 1-й переменной
cathetus2 = cathetus2 + cathetus2 * 0.1;     // модификация 2-й переменной
// последующие предложения
```

**Рис. 7.1.** Фрагмент кода, осуществляющего вычисление площадей прямоугольных треугольников

Затем следует *блок предложений*, состоящий из следующих предложений.

```
triangleArea = (cathetus1 * cathetus2)/2.0;
// запись значения triangleArea в таблицу
```

Первое предложение блока осуществляет вычисление значения площади первого прямоугольного треугольника и запись вычисленной площади в переменную `triangleArea`, а второе — запись значения `triangleArea` в таблицу (это действие, в коде, описано комментарием). Назовем перечисленные действия *выполнением блока*.

После выполнения блока осуществляется *модификация параметров цикла* предложениями.

```
cathetus1 = cathetus1 + cathetus1 * 0.1;
cathetus2 = cathetus2 + cathetus2 * 0.1;
```

Ясно, что способ многократного выполнения блока предложений, воплощенный в линейном коде на рис. 7.1, не является рациональным. Это становится особенно очевидным, если представить, что нам необходимо вычислить площади не пяти, а тысячи прямоугольных треугольников. Более рациональный способ заключается в таком способе записи кода, который на русском языке означает, примерно, следующее: «После начальной инициализации параметров многократно выполняй блок предложений и модифицируй параметры до тех пор, пока истинным остается некоторое условие». Такой способ записи кода называется циклом.

Теперь мы можем перечислить основные действия, из которых строится цикл. Представим их следующим списком:

- инициализация параметров цикла;
- модификация параметров цикла;
- выполнение блока предложений;
- проверка условия окончания итераций.

Отметим, что приведенный список следует понимать только как перечень действий, необходимых для организации цикла, а не как последовательность их выполнения.

Моделировать цикл удобно при помощи диаграммы деятельности, которую мы ранее использовали для моделирования ветвлений.

Как было отмечено, поведение программной системы, особенно в тех случаях, когда имеет место нарушение естественного порядка следования предложений, удобно представлять графически, при помощи диаграммы деятельности унифицированного языка моделирования UML. Диаграмма деятельности моделирует поведение в виде последовательно-параллельных потоков деятельностей, каждый из которых представляет собой последовательности шагов-деятельностей, приводящих к достижению одного или нескольких целевых состояний. Деятельность понимается в широком смысле. Это либо выполнение отдельного предложения, либо выполнение блока предложений, либо выполнение отдельного метода, либо выполнение группы логически связанных методов.

Начало диаграммы деятельности обозначается *стартовым символом*, представляющим собой жирную точку, а завершение диаграммы — *символом завершения*, который представляет собой жирную точку, изображенную внутри окружности, и соответствует некоторому целевому состоянию. Графическим *символом деятельности* является прямоугольник с закругленными углами, а сама диаграмма деятельности представляет собой множество символов деятельности, связанных между собой *символами переходов* от предыдущей деятельности к

последующей. Как правило, каждый графический символ деятельности, моделирующий блок предложений, порождает пару фигурных скобок в программном коде. Графический символ перехода, или ветвь, изображается в виде обыкновенной стрелки, исходящей из символа предыдущей деятельности и входящей в символ последующей деятельность.

Кроме графических символов деятельности и перехода, а также символов начала и завершения, диаграмма деятельности использует графические *символы ветвления* и *соединения*, изображаемые в виде ромбов. Символ ветвления имеет одну входящую ветвь и две исходящие ветви. Количество символов ветвления должно быть равно количеству символов соединения.

На рис. 7.2 приведены две диаграммы деятельности, моделирующие возможные логические варианты организации цикла. Модели цикла, приведенные на рис. 6.2, отличаются тем, в какой последовательности выполняются действия в итерационной части. Как это видно на рис. 7.2, обе модели предполагают, что перед выполнением итерационной части необходимо осуществить начальную инициализацию параметров цикла.

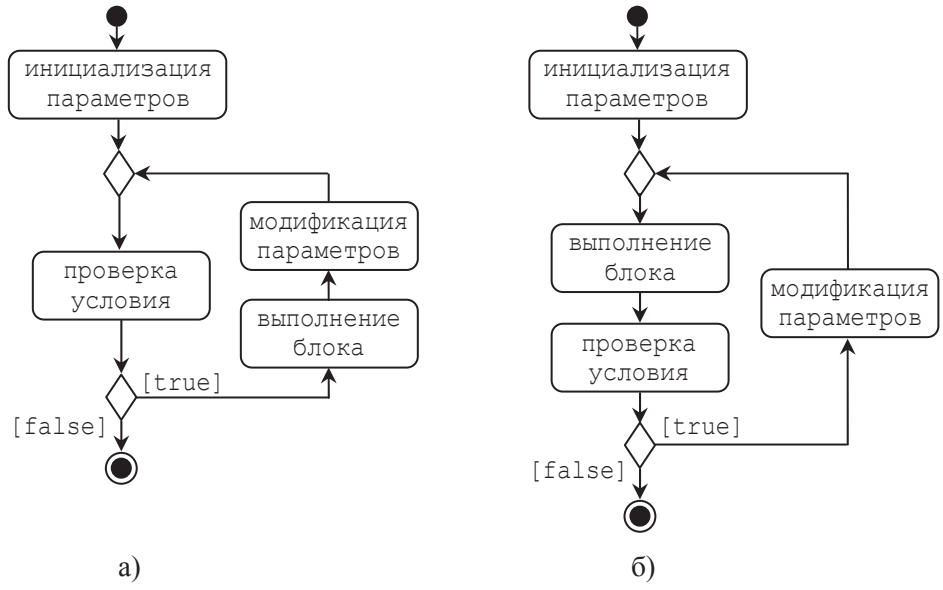



**Рис. 7.2.** Графическое представление цикла при помощи диаграммы деятельности.
а) цикл «вначале проверка условия, затем итерация»;
б) цикл «вначале итерация, затем проверка условия»

Цикл, изображенный в левой части рис. 7.2, организован таким образом, что в итерационной части цикла вначале осуществляется проверка условия окончания итераций, и только потом выполнение блока предложений. Ясно, что если сразу же после инициализации параметров проверка условия окончания итераций покажет, что оно не выполняется (направление `[false]` на диаграмме), то

итерационный процесс завершается, так ни разу и не выполнив блока предложений. Это является отличительной характеристикой цикла типа «вначале проверка условия, затем итерация».

Цикл, изображенный в правой части рис. 7.2, организован таким образом, что в итерационной части вначале осуществляется выполнение блока предложений, а затем проверка условия окончания итераций. Ясно, что при такой организации цикла блок предложений обязательно выполнится хотя бы один раз, даже в том случае, если сразу же после инициализации параметров условие не выполняется. Это является отличительной характеристикой цикла «вначале итерация, затем проверка условия».

## 7.2. Операторы цикла

В структуре кода, реализующего любую из моделей цикла, приведенных на рис. 7.2, часто выделяют заголовок цикла и тело цикла.

*Заголовок цикла* содержит оператор цикла, который служит для специфицирования условия завершения итераций, а в некоторых случаях для специфицирования параметров цикла и выражений, осуществляющих модификацию параметров.

*Тело цикла* представляет собой блок предложений, подлежащих многократному выполнению, и в некоторых случаях предложения, осуществляющие модификацию параметров цикла. В языке программирования Java имеется несколько операторов, используемых для кодирования циклов. Для кодирования цикла типа «вначале проверка условия, затем итерация» предназначены операторы `while` и `for`, а для кодирования цикла типа «вначале итерация, затем проверка условия» — оператор `do-while`.

### 7.2.1. Оператор `while`

Структура цикла на основе оператора `while`, приведена на рис. 7.3.

```
<инициализация параметров цикла>
while(<условие окончания итераций>)    // заголовок цикла
   {
    <блок предложений>                 // тело
    <модификация параметров цикла>     // цикла
   }
<последующие предложения>
```

**Рис. 7.3.** Структура цикла на основе оператора `while`

В заголовке цикла вначале записывается служебное слово `while`, за которым в круглых скобках размещается булево выражение, формулирующее условие окончания итераций.

Тело цикла, состоящее из некоторого количества предложений в фигурных скобках, помещается сразу же за заголовком цикла. Если тело цикла состоит из одного предложения, то фигурные скобки могут быть опущены.

В качестве примера, иллюстрирующего организацию цикла с помощью оператора `while`, разработаем код для решения задачи вычисления площадей ряда прямоугольных треугольников, сформулированной в начале настоящего раздела. На рис. 7.4. приведен фрагмент требуемого кода.

```
double cathetus1 = 3.0,       // объявление и инициализация 1-го параметра
       cathetus2 = 4.0;       // объявление и инициализация 2-го параметра
int counter = 1;              // объявление и инициализация 3-го параметра
float triangleArea = 0.0;
while(counter <= 5)           // заголовок цикла
   {
    triangleArea = (cathetus1 * cathetus2)/2.0;
    // запись значения triangleArea в таблицу
    cathetus1 = cathetus1 + cathetus1 * 0.1; // модификация 1-го параметра
    cathetus2 = cathetus2 + cathetus2 * 0.1; // модификация 2-го параметра
    counter++;                               // модификация 3-го параметра
   }
```

**Рис. 7.4.** Фрагмент кода, иллюстрирующий организацию цикла с помощью оператора `while`. Количество итераций заранее известно

Параметрами цикла являются оба катета `cathetus1` и `cathetus2`, а также целочисленная переменная `counter` (счетчик), предназначенная для подсчета количества итераций. Эти параметры объявлены и проинициализированы при помощи предложений, размещенных перед заголовком цикла. Перед заголовком цикла также объявлена и проинициализирована переменная `triangleArea`, в которую на каждой итерации будет записываться площадь треугольника.

В заголовке цикла записано булево выражение `counter <= 5`, которое определяет момент окончания итераций. Итерации продолжаются до тех пор, пока значение этого выражения истинно.

В теле цикла вначале вычисляется площадь прямоугольного треугольника для текущего значения катетов, а затем это значение записывается в таблицу (описано комментарием). Последние три предложения тела цикла осуществляют модификацию параметров в соответствии с условием задачи.

В приведенном примере цикла количество итераций известно заранее. Однако, оператор цикла `while` чаще всего используется в тех случаях, когда строится цикл

типа «вначале проверка условия, затем итерация» и программисту заранее неизвестно количество итераций.

Пусть, например, нам необходимо последовательно вычислять площади ряда прямоугольных треугольников, каждый раз увеличивая катеты предыдущего треугольника на 10%, а процесс вычислений продолжать до тех пор, пока площадь очередного треугольника не станет больше наперед заданной величины. Ясно, что количество итераций в этом случае заранее неизвестно, поскольку неизвестна наперед заданная граничная площадь. На рис. 7.5 приведен фрагмент кода, решающего эту задачу.

```
double cathetus1 = 3.0,       // объявление и инициализация 1-го параметра
       cathetus2 = 4.0,       // объявление и инициализация 2-го параметра
       triangleArea = 0.0,
       limitArea;
// инициализация limitArea
while(triangleArea < limitArea)              // заголовок цикла
   {
    triangleArea = (cathetus1 * cathetus2)/2.0;
    // запись значения triangleArea в таблицу
    cathetus1 = cathetus1 + cathetus1 * 0.1; // модификация 1-го параметра
    cathetus2 = cathetus2 + cathetus2 * 0.1; // модификация 2-го параметра
   }
```

**Рис. 7.5.** Фрагмент кода, иллюстрирующий организацию цикла с помощью оператора `while`. Количество итераций заранее неизвестно

Код, приведенный на рис. 7.5, отличается от кода на рис 7.4 тем, что в нем отсутствует параметр `counter` и объявлена новая переменная `limitArea` (граничная площадь) для хранения граничной площади треугольника. Изменено также булево выражение в заголовке цикла. Теперь оно имеет вид `triangleArea < limitArea` и, следовательно, равно значению `true` до тех пор, пока площадь текущего треугольника меньше, чем граничная площадь.

### 7.2.2. Оператор `for`

В тех случаях, когда количество итераций точно известно и цикл относится к типу «вначале проверка условия, затем итерация», удобно использовать оператор `for`, поскольку в заголовке этого оператора можно специфицировать все компоненты цикла, за исключением блока, подвергающегося итерациям. Структура цикла на основе оператора `for` приведена на рис. 7.6.

```
for(<параметры с нач. знач.>;<усл. окончания>;<выраж. модификации>)
   {
   <блок предложений>
   }
<последующие предложения>
```

**Рис. 7.6.** Структура цикла на основе оператора `for`

В заголовке цикла после служебного слова `for` и в круглых скобках последовательно размещаются: (1) предложения, объявляющие и инициализирующие параметры цикла; (2) булево выражение, формулирующее условие окончания итераций; и (3) выражения, осуществляющие модификацию параметра цикла. Разделителем между элементами заголовка в круглых скобках является точка с запятой. При кодировании цикла на основе оператора `for` с одним параметром последний выполняет функции счетчика итераций. Отметим, что если параметры цикла объявляются в заголовке цикла, то область видимости параметров ограничивается только телом цикла. За пределами тела цикла параметры являются неизвестными переменными.

Как и в случае цикла на основе оператора `while`, тело цикла, состоящее из некоторого количества предложений в фигурных скобках, помещается сразу же за заголовком цикла. Если тело цикла состоит из одного предложения, то фигурные скобки могут быть опущены.

На рис. 7.7 приведен фрагмент кода, иллюстрирующего цикл с одним параметром на основе оператора `for`. Код позволяет найти сумму целых положительных чисел в диапазоне от 1 до 1000.

```
int sum = 0;
for(int i = 1; i <= 1000; i++)
  sum += i;
```

**Рис. 7.7.** Фрагмент кода, вычисляющего сумму целых положительных чисел от 1 до 1000

Перед заголовком цикла размещается предложение, при помощи которого объявляется и инициализируется целочисленная переменная с именем `sum` (сумма), предназначенная для накопления суммы целых положительных чисел.

В заголовке цикла специфицированы: параметр цикла, условие окончания итераций и предложение, модифицирующее параметр. Параметром является целочисленная переменная с именем `i`, которая инициализирована значением 1. Условие окончания итераций определяется булевым выражением, которое останавливает итерации, если значение параметра цикла превысит 1000. Выражение, модифицирующее параметр цикла, увеличивает на единицу значение параметра цикла при каждой итерации.

Тело цикла состоит из предложения, которое на каждой итерации увеличивает значения переменной `sum` на величину текущего значения параметра цикла. Поскольку тело цикла состоит из одного предложения, то фигурные скобки опущены.

Цикл на основе оператора `for` можно использовать для нахождения степени вещественного числа для случая, когда показателем степени является натуральное число. Действительно, для возведения значения некоторой переменной с именем `base` (основание) в степень с показателем степени, который определяется значением переменной `exponent` (показатель степени), необходимо умножить значение переменной `base` само на себя `exponent` раз. Это можно сделать в цикле с количеством итераций, равным значению переменной `exponent`. На рис. 7.8 приведен фрагмент кода, осуществляющего возведение вещественного числа в натуральную степень.

```
double base,                          // число, возводимое в степень
       power = 1.0;                   // степень числа с начальным значением
int exponent;                         // показатель степени
// инициализация base и exponent
for(int i = 1; i <= exponent; i++) // заголовок цикла
   power *= base;                     // тело цикла
```

**Рис. 7.8.** Фрагмент кода для возведения вещественного числа в целочисленную степень

Перед заголовком цикла объявляются переменные, необходимые для кодирования: `base` (основание степени), `exponent` (показатель степени) и `power` (степень числа). Для правильной работы программы переменная `power` должна иметь начальное значение, равное единице. Затем тем или иным способом задаются значения числа, возводимого в степень (переменная `base`), и показателя степени (переменная `exponent`). Например, путем ввода значения при помощи клавиатуры. Цикл строится на основе оператора `for`. В заголовке объявляются: (1) целочисленная переменная с именем `i`, являющаяся параметром цикла; (2) выражение-условие окончания итераций `i <= exponent` и (3) выражение `i++`, осуществляющее модификацию параметра цикла. Заголовок обеспечивает ровно `exponent` итераций тела цикла. Тело цикла состоит из одного предложения и поэтому не заключено в фигурные скобки. При каждой итерации предыдущее значение переменной `power` умножается на значение переменной `base` и полученное произведение опять записывается в переменную `power`.

На рис. 7.9 приведен код, иллюстрирующий цикл на основе оператора `for`, в котором используется несколько параметров.

```
int sumForward = 0,
    sumBackward = 0;
for(int i = 1, j = 1000; i < j; i++, j--)
   {
    sumForward += i;
    sumBackward += j;
   }
```

**Рис. 7.9.** Цикл на основе оператора `for` с двумя параметрами цикла

Перед началом цикла объявляются и инициализируются нулями две целочисленные переменные: `sumForward` (сумма вперед) и `sumBackward` (сумма назад). Первый компонент заголовка объявляет и инициализирует два параметра цикла с именами `i` и `j`. Последний компонент заголовка содержит два выражения модификации этих параметров. Перед началом итераций параметр `i` принимает значение 1, а параметр `j` — значение 1000. При каждой итерации параметр `i` увеличивается на 1, а параметр `j` уменьшается на единицу. Как видно на рис. 7.9, в том случае, когда в заголовке описывается несколько параметров, то элементы описания разделяются запятыми. Тело цикла состоит из двух предложений. Предложение `sumForward += i` в процессе итераций суммирует натуральные числа в прямом направлении, начиная с числа 1, а предложение `sumBackward += j` — в обратном направлении, начиная с числа 1000. Итерационный процесс продолжается до тех пор, пока значение первого параметра меньше, чем значение второго.

Структура цикла на основе оператора `for`, приведенная на рис. 7.6, не является фиксированной. Язык программирования Java допускает различные варианты кодирования цикла на основе оператора `for`. Например, в рассмотренных ранее случаях всегда предполагалось, что параметр или параметры цикла и объявляются, и инициализируются в его заголовке. Однако это не обязательно. Параметры могут быть объявлены перед заголовком цикла, а в заголовке только проинициализированы. Поэтому код, приведенный на рис. 7.9, может быть переписан так, как это показано на рис. 7.10.

```
int sumForward = 0, sumBackward = 0;
int i,j;                                  // объявление параметров
for(i = 1, j = 1000; i < j; i++, j--){
    sumForward += i;
    sumBackward += j;
   }
```

**Рис. 7.10.** Внешнее объявление параметров цикла на основе оператора `for`

Если параметры цикла объявлены перед заголовком цикла, то их нельзя переопределять в заголовке. Например приведенные ниже строки кода вызовут ошибку на этапе компиляции.

```
int i,j;
for(int i = 1, j = 1000; i < j; i++, j--)
```

Оба оператора `while` и `for` позволяют строить циклы типа «вначале проверка условия, затем выполнение блока», однако оператор `while` является более универсальным. Любой цикл на основе оператора `for` может быть перекодирован в цикл на основе оператора `while`, однако обратное не всегда возможно.

### 7.2.3. Оператор do-while

Оператор do-while применяется при кодировании программы с циклом типа «вначале итерация, затем проверка условия». Структура цикла на основе оператора do-while приведена на рис. 7.11.

```
<инициализация параметров цикла>
do{
   <блок предложений>                    // тело
   <модификация параметров цикла>        // цикла
  }
while (<условие окончания итераций>)     // заголовок цикла
<последующие предложения>
```

**Рис. 7.11.** Структура цикла на основе оператора do-while

В цикле типа «вначале итерация, затем проверка условия» после объявления и начальной инициализации переменных цикла пишется служебное слово do, после которого размещается тело цикла, состоящее из блока предложений, включающих предложения модификации параметров цикла. Заголовок цикла, начинающийся со служебного слова while, помещается после тела цикла. Поэтому при каждой итерации вначале выполняется тело цикла, а затем проверяется условие окончания итераций. Если условие истинно, то итерация повторяется. В противном случае итерации завершаются.

Оператор do-while часто применяется при кодировании программы, обслуживающей меню. Меню состоит из некоторого количества пунктов, предлагаемых пользователю. С каждым пунктом связаны определенные действия программы.

Основной частью кода программы, обслуживающей меню, является цикл, поскольку пользователь меню может осуществлять выбор пунктов меню сколь угодно много раз. В теле цикла на каждой итерации детерминируется выбранный пункт меню, и управление передается соответствующему коду-обработчику выбранного пункта. После отображения меню на мониторе пользователь осуществляет выбор пункта меню. Следовательно, в цикле, обслуживающем меню, первая итерация должна быть выполнена еще до проверки условия выхода из цикла.

На рис. 7.12 приведен фрагмент кода, иллюстрирующий организацию цикла на основе оператора do-while. Код обслуживает меню, состоящее из трех пронумерованных пунктов. Выбор пункта меню осуществляется вводом его номера.

Перед циклической частью кода на монитор выводится меню, состоящее из заголовка меню и его пунктов. Затем объявляется параметр цикла с именем choice (выбор). Спецификой цикла на основе оператора do-while, обслуживающего меню, является то, что параметр цикла только объявляется, но не инициализируется перед входом в тело цикла. Инициализация параметра осуществляется первым действием тела цикла при первой итерации. При последующих итерациях это действие осуществляет модификацию параметра.

После инициализации/модификации параметра цикла (переменной `choice`) управление передается одному из обработчиков пунктов меню. Вызов обработчика, в соответствии с введенным номером пункта меню, осуществляется кодом на основе оператора `switch`. В этот кода не включена `default`-структура, поскольку правильность ввода номера пункта меню проверяется булевым выражением заголовка цикла.

```
// вывод пронумерованных пунктов меню;
int choice;
do {
   // ввод номера пункта меню, инициализация/модификация choice
   switch (choice){
      case 1:
      // вызов обработчика пункта меню 1
      break;
      case 2:
      // вызов обработчика пункта меню 2
      break;
      case 3:
      // вызов обработчика пункта меню 3
      break;
   }
}
while((choice < 1) || (choice > 3));
// вывод сообщения об ошибочном вводе номера пункта меню
```

**Рис. 7.12.** Фрагмент кода, иллюстрирующий организацию цикла на основе оператор `do-while`

После обработки пункта меню проверяется условие окончания итераций. Цикл будет повторяться сколь угодно много раз, пока пользователь меню вводит правильные номера пунктов меню (числа 1, 2 или 3).

Код, обслуживающий меню и приведенный на рис. 7.12, обладает тем недостатком, что имеется только один способ завершения итераций. Для завершения итераций и выхода из меню необходимо ввести «неправильный» пункт меню, например, число 0 или 4.

## 7.3. Вложенные циклы

Блок предложений, который подвергается итерациям в цикле, не обязательно имеет линейную структуру. Он может включать одно или несколько операторов, нарушающих естественный порядок следования предложений. Код,

приведенный на рис. 7.12, иллюстрирует такую возможность. В тело цикла включен оператор `switch`.

Операторы `while`, `for` и `do-while`, так же, как и изученные ранее операторы `if` и `switch`, относятся к операторам, нарушающим естественный порядок выполнения предложений программного кода, и могут быть включены в блок предложений любого цикла. Вложенные циклы образуются, если тело некоторого (внешнего) цикла содержит другой (внутренний) цикл.

При классификации вложенных циклов необходимо учитывать два классификационных признака. Во-первых, типы внешнего и внутреннего циклов в соответствии с моделями, приведенными на рис. 7.2 (цикл типа «вначале проверка условия, затем итерация» и цикл типа «вначале итерация, затем проверка условия»). Во-вторых, типы операторов, используемые для организации внешнего и внутреннего циклов (`while`, `for` и `do-while`). Если ограничиться глубиной вложения равной двум, когда имеется только один внешний и один внутренний циклы, то первый классификационный признак порождает четыре модели вложенных циклов.

1. Внешний цикл типа «вначале проверка, затем итерация», внутренний цикл «вначале проверка, затем итерация».
2. Внешний цикл типа «вначале проверка, затем итерация», внутренний цикл «вначале итерация, затем проверка».
3. Внешний цикл типа «вначале итерация, затем проверка», внутренний цикл типа «вначале проверка, затем итерация».
4. Внешний цикл типа «вначале итерация, затем проверка», внутренний цикл типа «вначале итерация, затем проверка».

На рис. 7.13 приведена диаграмма деятельности, соответствующая первой модели, когда в итерируемый блок внешнего цикла типа «вначале проверка, затем итерация» вкладывается внутренний цикл такого же типа.

Из модели, приведенной на рис. 7.13, однозначно следует, что *в процессе выполнения одной итерации внешнего цикла выполняются все итерации внутреннего цикла*. Таким образом, общее количество итераций вложенных циклов, модель которых приведена на рис. 7.13, равно произведению количества итераций внешнего цикла на количество итераций внутреннего цикла.

Для организации цикла типа «вначале проверка условия, затем итерация» могут использоваться оба оператора `while` и `for`. В зависимости от комбинации этих операторов, во внешнем и внутреннем циклах возможны следующие четыре варианта отображения модели, приведенной на рис. 7.13, в программный код:

- внешний цикл на основе `while`, внутренний цикл на основе `while`;
- внешний цикл на основе `while`, внутренний цикл на основе `for`;
- внешний цикл на основе `for`, внутренний цикл на основе `while`;
- внешний цикл на основе `for`, внутренний цикл на основе `for`.

На рис. 7.14 приведена структура кода, соответствующая модели вложенных циклов, приведенной на рис. 7.13, для случая, когда для кодирования и внешнего и внутреннего циклов используется оператор `while`.

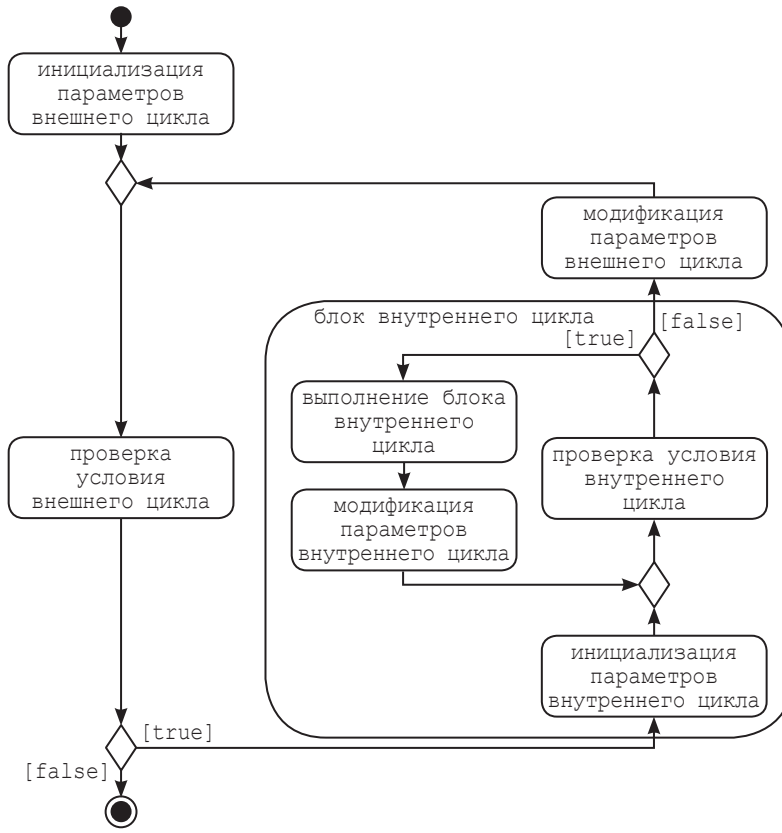



**Рис. 7.13.** Модель вложенных циклов. Во внешний цикл типа «вначале проверка условия, затем итерация» вкладывается внутренний цикл такого же типа

```
<инициализация параметров внешнего цикла>
while (<условие окончания итераций внешнего цикла>)        // внешний
   {
    <блок внешнего цикла>
    <инициализация параметров внутреннего цикла>
    while (<условие окончания итераций внутреннего цикла>) // внутренний
       {
        <блок внутреннего цикла>
        <модификация параметров внутреннего цикла>
       }
    <модификация параметров внешнего цикла>
   }
<последующие предложения>
```

**Рис. 7.14.** Структура вложенных циклов, соответствующая модели на рис. 7.13. Внешний и внутренний циклы организованы на основе оператора `while`

Рассмотрим следующий пример. Пусть нашей задачей является разработка кода, который формирует и выводит на экран монитора десять таблиц умножения целых чисел от 1 до 10. Фрагмент кода, решающий эту задачу, приведен на рис. 7.15.

```
int product;
int multiplr1 = 1;
while (multiplr1 <= 10)
   {
    int multiplr2 = 1,
        product = 0;
        System.out.println("Таблица умножения для "+ multiplr1);
    while (multiplr2 <= 10)
       {
        product = multiplr1 * multiplr2;
        System.out.println(multiplr1 + " * " + multiplr2 + " = " + product);
        multiplr2++;
       }
    multiplr1++;
    System.out.println();
   }
```

**Рис. 7.15.** Фрагмент кода, иллюстрирующий вложенные циклы со структурой, приведенной на рис. 7.14

Первое предложение объявляет и инициализирует переменную с именем `multiplr1` (сомножитель 1), являющуюся параметром внешнего цикла. Затем следует заголовок внешнего цикла с условием окончания итераций внешнего цикла `multiplr1 <= 10`. Согласно этому условию, итерации тела внешнего цикла будут продолжаться до тех пор, пока первый сомножитель меньше числа 10. Первое предложение тела внешнего цикла объявляет и инициализирует две переменные: переменную с именем `multiplr2` (сомножитель 2), являющуюся параметром внутреннего цикла, и переменную с именем `product` (произведение), предназначенную для хранения произведения сомножителей. Затем на экран монитора выводится заголовок текущей таблицы умножения.

Заголовок внутреннего цикла специфицирует условие окончания итераций внутреннего цикла в виде `multiplr1 <= 10`, которое останавливает итерации, если второй сомножитель становится б*о*льшим, чем число 10. Тело внутреннего цикла начинается предложением, при помощи которого находится произведение сомножителей, за которым следует предложение, выводящее текущую строку текущей таблицы умножения. Затем модифицируется параметр внутреннего цикла (его значение увеличивается на единицу), и этим завершается тело внутреннего цикла.

После внутреннего цикла размещены два предложения тела внешнего цикла. Первое из них модифицирует параметр внешнего цикла, увеличивая его на единицу, а второе выводит на экран монитора пустую строку. В результате работы программы на экране будут последовательно выведены таблицы умножения в виде.

```
Таблица умножения для 1
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
1 * 6 = 6
1 * 7 = 7
1 * 8 = 8
1 * 9 = 9
1 * 10 = 10

Таблица умножения для 2
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
2 * 10 = 20

. . .
```

Рассмотрим теперь структуру кода, которая соответствует модели вложенных циклов, приведенной на рис. 7.13, для случая, когда для кодирования и внешнего, и внутреннего циклов используется оператор `for`. Эта структура приведена на рис. 7.16.

```
for(<параметры внеш.>; <условие внеш.>; <модификация внеш.>)
   {
    <блок внешнего цикла>
       for(<параметры внутр.>; <условие внутр.>; <модификация внутр.>)
          {
           <блок внутреннего цикла>
          }
   }
<последующие предложения>
```

**Рис. 7.16.** Структура вложенных циклов, соответствующая модели на рис. 7.13. Внешний и внутренний циклы организованы на основе оператора `for`

Для кодирования задачи формирования и последовательного вывода на экран монитора десяти таблиц умножения для целых чисел от 1 до 10 (см. рис. 7.15) удобно использовать вложенные циклы не на основе оператора `while`, а на основе оператора `for`, поскольку количество итераций как для внешнего, так и для внутреннего цикла в точности известно заранее.

На рис. 7.17 приведен фрагмент кода, решающий задачу построения таблицы умножения на основе структуры вложенных циклов на рис. 7.16.

```
int product;
for (int multiplr1 = 1; multiplr1 <= 10; multiplr1++)
   {
    product = 0;
    System.out.println("Таблица умножения для "+ multiplr1);
    for (int multiplr2 = 1; multiplr2 <= 10; multiplr2++)
       {
        product = multiplr1 * multiplr2;
        System.out.println(multiplr1 + " * " + multiplr2 + " = " + product);
       }
    System.out.println("");
   }
```

**Рис. 7.17.** Фрагмент кода, иллюстрирующий вложенные циклы со структурой, приведенной на рис. 7.16

Фрагмент кода на рис. 7.17 компактнее, чем на рис. 7.15, поскольку почти все компоненты (объявление и инициализация параметров, условия окончания итераций и модификация параметров) как внешнего, так и внутреннего циклов специфицированы в их заголовках. Тела циклов содержат только блоки итерируемых предложений.

## 7.4. Оператор `break`

Оператор `break` используется для прерывания работы блока предложений. Точкой прерывания является предложение с оператором `break`. Существуют три типовых варианта использования оператора `break` в программном коде:

- для безусловного прерывания работы блока;
- для условного прерывания работы блока;
- для условного прерывания работы помеченного блока.

В случае безусловного прерывания работы блока предложений с помощью оператора `break`, структура программного кода имеет вид, приведенный на рис. 7.18.

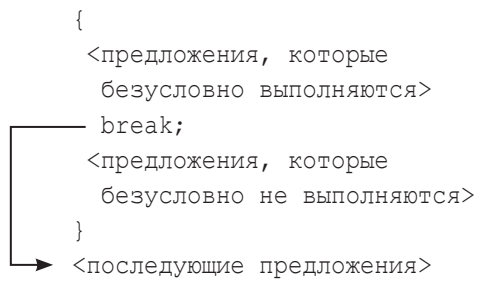



**Рис. 7.18.** Структура кода с оператором `break`, который выполняет безусловное прерывание работы блока

Оператор `break`, размещенный внутри блока так, как это показано на рис. 7.18, выполняет безусловное прерывание работы блока. Предложения, записанные до оператора `break`, безусловно выполняются, а предложения, записанные после оператора `break`, безусловно не выполняются. Оператор `break` передает управление первому предложению, непосредственно следующему за блоком. В том случае, когда блок с оператором `break` вложен в другой блок, то прерыванию подвергается работа только внутреннего блока. Например, оператор `break`, размещенный внутри блока внутреннего цикла, прерывает работу только внутреннего цикла и не оказывает влияния на внешний цикл.

Ранее мы познакомились с примерами использования оператора `break` для безусловного прерывания работы оператора `switch` (см. подраздел 6.4). В том случае, когда произошел выбор одной из `case`-структур оператора `switch` и успешно выполнился блок этой `case`-структуры, оператор `break` осуществляет безусловное прерывание работы оператора `switch`.

В случае условного прерывания работы блока предложений при помощи оператора `break`, структура программного кода имеет вид, приведенный на рис. 7.19.

Для организации условного прерывания работы блока оператор `break` размещается в предложении с оператором `if`.

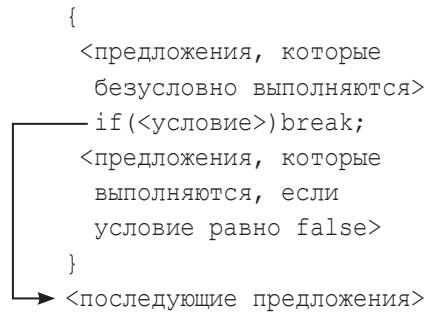



**Рис. 7.19.** Структура кода с оператором `break`, который выполняет условное прерывание работы блока

Предложения блока, записанные до предложения с оператором `break`, безусловно выполняются, а предложения, записанные после условного оператора `break`, выполняются только в том случае, если условие, специфицированное в операторе `if`, ложно. Если условие истинно, то работа блока прерывается и управление передается первому предложению, непосредственно следующему за блоком. Если имеет место вложение одного блока в другой, то условное прерывание касается только внутреннего блока.

Рассмотрим пример использования условного оператора `break` для выхода из «бесконечного цикла». Задача заключается в разработке кода, позволяющего пользователю сколь угодно много раз вводить пароль. Код завершает работу только в том случае, если пользователь ввел правильный пароль. Если пользователь ввел неправильный пароль, то ему предлагается совершить новую попытку. На рис. 7.20 приведен фрагмент кода, решающий сформулированную задачу.

```
String rightPassword = "a4b3c2d1";
String password;
while(true)
   {
    System.out.println("Введите пароль");
   // ввод пароля в password
   if(password == rightPassword) break;
   }
```

**Рис. 7.20.** Фрагмент кода, иллюстрирующий использование условного оператора `break` для выхода из «бесконечного» цикла

Переменная `rightPassword` хранит правильный пароль, а переменная `password` предназначена для хранения пароля, вводимого пользователем. После объявления обеих переменных и инициализации переменной `rightPassword` организован «вечный» цикл при помощи оператора `while`, у которого условие окончания итераций всегда истинно.

В теле цикла многократно выполняются три предложения: (1) предложение, выводящее на экран монитора приглашение ввести пароль; (2) предложение, присваивающее введенный пользователем пароль переменной `password` (отмечено комментарием) и (3) предложение `if(password == rightPassword) break`. Последнее предложение осуществляет прерывание работы цикла в том случае, когда пользователь вводит правильный пароль.

Как было отмечено выше, если оператор `break` выполняет безусловное или условное прерывание внутреннего цикла, то завершается работа только внутреннего цикла. На работу внешнего цикла действие этого оператора не распространяется. Рассмотрим пример, иллюстрирующий отмеченное локальное действие условного оператора `break` на вложенные циклы.

На рис. 7.21 приведен фрагмент кода с вложенными циклами для случая, когда оба цикла организованы на основе оператора `for`. Код осуществляет вывод на экран

монитора десяти строк, каждая из которых состоит из некоторого количества символов «звездочка».

```
for(int i=1; i<=10; i++)
   {
   for(int j=1; j<=10; j++)
      {
       System.out.print("*");
       if(i==j) break;
      }
   System.out.println();
   }
```

**Рис. 7.21.** Фрагмент кода, иллюстрирующий локальное действие оператора `break` на вложенные циклы

Внешний цикл, с параметром `i`, управляет выводом строк, а внутренний, с параметром `j`, — выводом отдельной строки.

Условный оператор `break` прерывает итерации внутреннего цикла в том случае, когда имеет место равенство параметров внешнего и внутреннего циклов. Это эквивалентно выводу в строку количества символов, равного номеру строки. Таким образом, введение условного оператора `break` во внутренний цикл приводит к тому, что количество итераций внутреннего цикла становится переменной величиной. При первом выполнении внутреннего цикла осуществляется одна итерация, а при каждом повторном — количество итераций увеличивается на единицу. Количество же итераций внешнего цикла остается неизменным и определяется его заголовком. В результате работы программы на экран будут выведены следующие строки символов.

```
*
**
***
****
*****
******
*******
********
*********
**********
```

Правило, согласно которому условный оператор `break`, размещенный во внутреннем блоке, прерывает работу только внутреннего цикла, ограничивает возможности кодирования. Встречаются ситуации, когда при нарушении некоторого

условия необходимо прервать работу не только внутреннего, но и внешнего циклов. Такое прерывание можно выполнить при помощи условного оператора `break` с меткой. На рис. 7.22 приведена структура программного кода, осуществляющего условное прерывание помеченного блока при помощи оператора `break` с меткой.

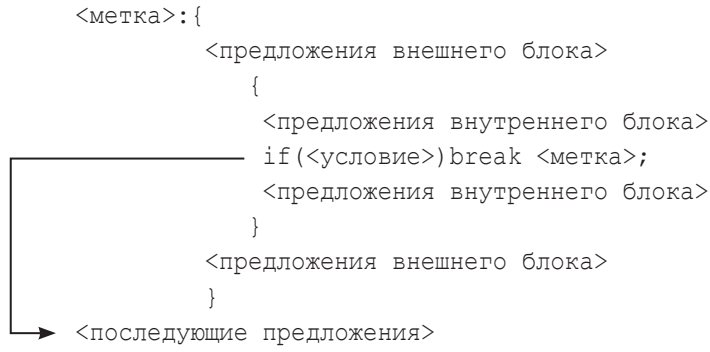



**Рис. 7.22.** Структура кода с оператором `break`, который выполняет условное прерывание блока, помеченного меткой

На рис. 7.23 приведен фрагмент кода, иллюстрирующий применение условного оператора `break` с меткой.

```
outer:for(int i = 1; i < 10; i++)
          {
           System.out.println("Внешний");
           for(int j = 1; j < 10; j++)
              {
               System.out.println("Внутренний");
               if(j == 4) break outer;
              }
           System.out.println("Опять внешний");  // не будет выведено
          }
System.out.println("Предложение после циклов");
```

**Рис. 7.23.** Фрагмент кода, иллюстрирующий применение условного оператора `break` с меткой

Код, приведенный на рис. 7.23, представляет собой вложенные циклы, организованные на основе оператора `for`. Заголовок внешнего цикла снабжен меткой с именем `outer`. Первое предложение в теле внешнего цикла выводит на экран слово «`Внешний`». Затем следует заголовок внутреннего цикла. Тело внутреннего цикла состоит из двух предложений. Первое предложение выводит на экран слово «`Внутренний`», а второе — представляет собой предложение, включающее оператор `if` и оператор `break` с меткой `outer`. Булево выражение оператора `if` принимает значение `true` в том случае, когда параметр внутреннего цикла с именем `j` становится

равным четырем. В этом случае оператор `break` прерывает работу обоих циклов и управление передается первому предложению, записанному непосредственно после тела внешнего цикла. Предложения внешнего цикла, размещенные после тела внутреннего цикла, не выполняются. После выполнения кода, приведенного на рис. 7.23, на экране должны появиться следующие строки.

```
Внешний
Внутренний
Внутренний
Внутренний
Внутренний
Предложение после циклов
```

Рассмотренные способы прерывания работы блока с помощью оператора `break` предназначены для организации прерывания работы блоков внутри метода и используются в практике императивного программирования при кодировании методов. Язык программирования Java поддерживает более общий механизм прерывания работы методов, предназначенный для использования на уровне объектно-ориентированного программирования. Это механизм исключительных ситуаций или исключений. Технология прерываний на основе исключительных ситуаций исходит из предположения, что работа программы осуществляется не в детерминированной (предсказуемой), а в стохастической среде, в которой случайным образом могут возникать исключительные ситуации. Исключительной ситуацией называется событие, препятствующее нормальной работе программного кода. Если при написании кода используется механизм исключительных ситуаций, то прерывание работы метода происходит в те моменты, когда возникает одна из исключительных ситуаций. Прерывание передает управление обработчику исключительной ситуации, соответствующему типу исключительной ситуации. Способы учета и моделирования исключительных ситуаций на этапе проектирования программы описаны во второй части пособия.

## 7.5. Оператор `continue`

Оператор `continue` применяется только при кодировании циклов. Оператор осуществляет прерывание текущей итерации и переход к следующей итерации.

Прерывание итерационного процесса можно выполнить как оператором `break`, так и оператором `continue`. Однако, это два различных вида прерываний. Оператор `break` полностью останавливает итерационный процесс и передает управление первому предложению, размещенному после цикла. Оператор `continue` прерывает только текущую итерацию, не останавливая итерационный процесс. Оператор `continue` применяется в тех случаях, когда в ходе итерационного процесса

необходимо пропустить и не выполнять итерации, удовлетворяющие некоторому условию. Существуют два типовых варианта использования оператора `continue` в программном коде:

- для условного прерывания текущей итерации в одном цикле;
- для условного прерывания текущей итерации во вложенных циклах.

В первом варианте используется оператор `continue` без метки, а во втором — оператор `continue` с меткой. Фрагмент кода на рис. 7.24 иллюстрирует использование оператора `continue` без метки.

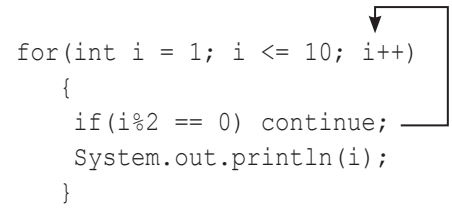



**Рис. 7.24.** Фрагмент кода, иллюстрирующий использование оператора `continue` без метки

Фрагмент кода на рис. 7.24 представляет собой цикл, организованный на основе оператора `for`. Параметром цикла является целочисленная переменная с именем `i`, которая изменяется в диапазоне от 1 до 10 с шагом 1.

Первое предложение тела цикла включает операторы `if` и `continue`. Булево выражение в операторе `if` проверяет, является ли текущее значение параметра цикла четным числом. Число является четным, если остаток от деления этого числа на два равен нулю.

Если параметр цикла — число четное, то выполнение текущей итерации прерывается, и цикл переходит к следующей итерации с модифицированным значением параметра. Поэтому следующим после оператора `continue` должно выполниться выражение, модифицирующее параметр цикла (стрелка на рис. 7.24). Если параметр — число нечетное, то оператор `continue` не выполняется и на экран выводится значение параметра. Таким образом, после завершения работы кода на экран будут выведены только нечетные значения переменной `i`, по одному значению в строке.

Если необходимо прервать текущие итерации вложенных циклов, то используется оператор `continue` с меткой. Оператор размещается в теле внутреннего цикла. Текущие итерации как внутреннего, так и внешнего циклов прерываются, и начинается новая итерация с внешнего цикла, помеченного той же меткой, которая указана в операторе `continue`. Фрагмент кода на рис. 7.25 иллюстрирует использование оператора `continue` с меткой.

Код на рис. 7.25 построен на основе двух циклов, вложенных друг в друга. И внешний и внутренний циклы организованы на основе оператора `for`. Параметром внешнего цикла является целочисленная переменная `i`, которая изменяется в пределах от 0 до 4 с шагом 1. Параметром внутреннего цикла является целочисленная переменная `j`, которая изменяется в тех же пределах, что и

параметр внешнего цикла. Код выводит на экран некоторое количество строк чисел. Числа в строке представляют собой произведения текущих значений параметров циклов. Внешний цикл управляет выводом строк, а внутренний — выводом отдельной строки.

```
outer:for(int i = 0; i < 4; i++) {
   for(int j = 0; j < 4; j++) {
      if(j > i) {
         System.out.println();
         continue outer;
         }
      System.out.print((i * j) + " ");
   }
System.out.println();
}
```

**Рис. 7.25.** Фрагмент кода, иллюстрирующий использование оператора `continue` с меткой

В начале тела внутреннего цикла расположен блок оператора `if`. Блок включает оператор `continue` с меткой `outer` (внешний). Булево выражение оператора `if` принимает значение `true` в тех случаях, когда значение параметра внутреннего цикла больше значения параметра внешнего цикла. В этих случаях оператор `continue` прерывает текущие итерации и передает управление выражению, модифицирующему параметр внешнего цикла. Если булево выражение оператора `if` принимает значение `false`, то вместо блока оператора `if` выполняется предложение внутреннего цикла `System.out.print((i * j) + " ")`, которое выводит на экран строку произведений параметров внешнего и внутреннего циклов, разделенных пробелом. Два предложения `System.out.println()` обеспечивают переход на новую строку при любых значениях булевого выражения оператора `if`.

Операторы `if` и `continue`, работая совместно, делают количество итераций внутреннего цикла переменной величиной, изменяющейся от 1 до 5. Первый раз выполняется одна итерация, затем две и т. д. Поскольку внутренний цикл формирует строку вывода, то количество чисел в строке также является переменной величиной и в каждой последующей строке увеличивается на единицу.

После завершения работы кода на экране будут сформированы следующие строки.

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
```

## 7.6. Итерационные процессы в математических алгоритмах

В основе большого количества математических алгоритмов лежат итерационные процессы. В ряде случаев математические алгоритмы предполагают такое большое количество итераций, что это делает невозможным их практическое использование без привлечения компьютеров. Компьютерные программы на основе однократных и вложенных циклов, автоматизируют итерационные вычисления. Производительность современных процессоров настолько велика, что даже очень большое количество итераций выполняется за доли секунды. Таким образом, умение кодировать итерационные процессы является необходимым условием умения трансформировать математические алгоритмы на основе итерационных процессов в практически пригодные компьютерные программы.

В настоящем подразделе рассмотрены примеры разработки программных кодов, реализующих некоторые простые математические алгоритмы, в основе которых лежат итерационные процессы:

- тестирование простоты натурального числа;
- алгоритмы Эвклида;
- вычисление факториала;
- вычисление чисел Фибоначчи;
- вычисление полинома по схеме Горнера;
- вычисление экспоненциальной функции при помощи ряда Тейлора.

### 7.6.1. Моделирование алгоритмов

Многие математические алгоритмы на основе итераций были придуманы задолго до изобретения компьютеров и программирования. В математической литературе они описаны на естественном языке с включением математических символов и формул.

Непосредственная трансляция описаний математических методов в программный код является ментально трудной задачей, которая чревата появлением в программном коде логических ошибок. Более рациональной является технология, которая предполагает двухэтапную трансляцию естественно-языкового описания алгоритма в программный код.

На первом этапе словесное описание алгоритма транслируется в модель на основе диаграммы деятельности. Модель представляет собой диаграмму, которая отражает пошаговую суть алгоритма и позволяет проследить потоки деятельностей от начальной деятельности до деятельности, приводящей к достижению цели. Графическое представление алгоритма позволяет легко проверить логику будущего кода.

На втором этапе диаграмма деятельности транслируется в код. Для этого достаточно знать, каким образом структурные элементы модели отображаются в программный код на том или ином языке программирования.

При построении модели целесообразно использовать не исходные графические символы-примитивы диаграммы деятельности (стартовый символ, символ деятельности, символ перехода, символ ветвления, символ соединения и другие), а готовые структуры, моделирующие набор стандартных типовых поведений, из которых строится любой алгоритм. К таким типовым поведенческим структурам относятся: блок предложений; модели выбора альтернативных блоков, приведенные на рис. 6.6, и модели циклов, приведенные на рис. 7.2. Назовем, для краткости, эти типовые структуры: (1) структура типа блок, (2) структура типа развилка и (3) структура типа цикл. Модели типовых поведенческих структур приведены на рис. 7.26.

Каждая из типовых поведенческих структур обладает только одни стартовым символом и только одним символом завершения. Из этого факта следуют два способа конструирования моделей алгоритмов из типовых поведенческих структур.

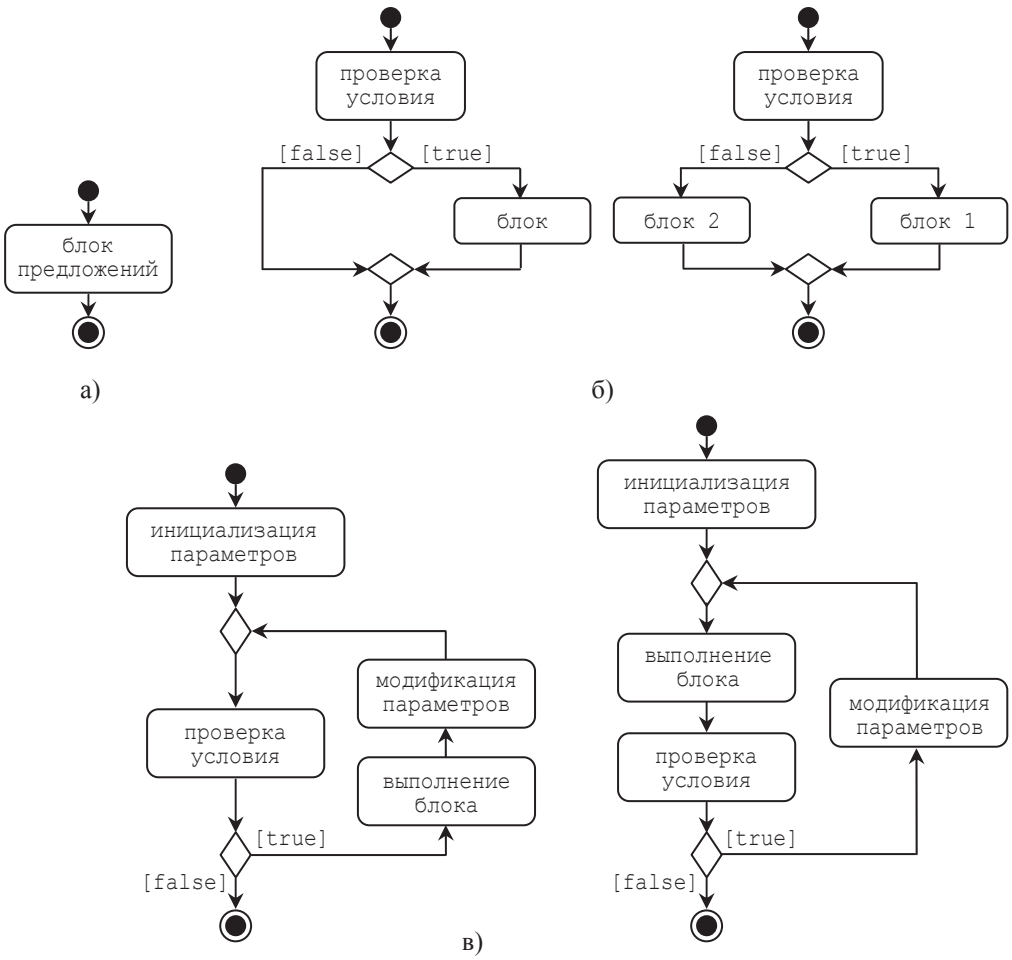



**Рис. 7.26.** Типовые поведенческие структуры алгоритмов. а) структура типа блок; б) варианты структуры типа развилка; в) варианты структуры типа цикл

Типовые структуры могут соединяться последовательно и в любой комбинации и, следовательно, первым способом конструирования моделей алгоритмов из типовых поведенческих структур является последовательное соединение типовых структур.

Второй способ конструирования моделей алгоритмов из типовых поведенческих структур нам уже известен — это вложение одной из типовых структур в блок другой типовой структуры. При вложении одной структуры в блок другой могут использоваться разнотипные структуры. Например, в один блок структуры типа развилка может вкладываться структура типа цикл, а в другой — еще одна структура типа развилка. Степень вложения, по крайней мере, умозрительно, не ограничена.

Комбинируя отмеченные способы конструирования моделей алгоритмов из типовых поведенческих структур, можно построить модели сколь угодно сложных алгоритмов.

Для комментирования элементов модели используется графический символ комментария, который представляет собой стилизованное изображение прямоугольного листа бумаги с загнутым верхним правым углом. Символ комментария соединяется с комментируемым элементом модели при помощи пунктирной линии.

### 7.6.2. Тестирование простоты натурального числа

Натуральное число $n$ называется *простым*, если оно имеет только два делителя: число 1 и число $n$ (делится без остатка только на единицу и на самого себя). Например, число 13 является простым, так как имеет два делителя: 1 и 13, а число 12 не является простым, так как делится без остатка на 1, 2, 3, 4, 6 и 12. Не простые числа (кроме числа 1) называются составными.

Алгоритм тестирования простоты должен отвечать на вопрос: «Является ли заданное натуральное число $n$ простым или составным?»

Существует несколько алгоритмов, осуществляющих тестирование простоты заданного натурального числа. Одним из наиболее простых является *алгоритм полного перебора возможных делителей*. Алгоритм основан на переборе всех натуральных чисел и обнаружении всех делителей тестируемого числа. Если оказывается, что делителями являются только число 1 и само тестируемое число, то последнее является простым. Если же находится хотя бы еще один делитель, то число является составным.

Обычно при переборе натуральных чисел в поисках делителей не участвует число 1, и перебор начинается с числа 2. Последним числом, участвующим в переборе, может быть число $n$–1. Однако известно, что значение наибольшего делителя для составного числа $n$ не может превышать величины $\sqrt{n}$. Поэтому, как правило, последним числом, участвующим в тестировании, является целая часть от $\sqrt{n}$.

Таким образом, тестирование на простоту натурального числа $n$ алгоритмом полного перебора основано на переборе всех натуральных чисел от числа 2 до целой части числа $\sqrt{n}$ и в вычислении остатка от деления $n$ на текущее натуральное число. Если остаток равен 0, то это означает, что текущее натуральное число

является делителем. Тогда число *n* объявляется составным, и алгоритм завершает работу. В противном случае выбирается следующее натуральное число, и алгоритм продолжает работу. Если выполнен перебор всех натуральных чисел от числа 2 до целой части $\sqrt{n}$ и не обнаружен ни один остаток, равный 0 (не обнаружен ни один делитель), то число *n* объявляется простым, и алгоритм завершает работу.

Транслируем словесное описание алгоритма полного перебора возможных делителей в диаграмму деятельности. Модель приведена на рис. 7.27.

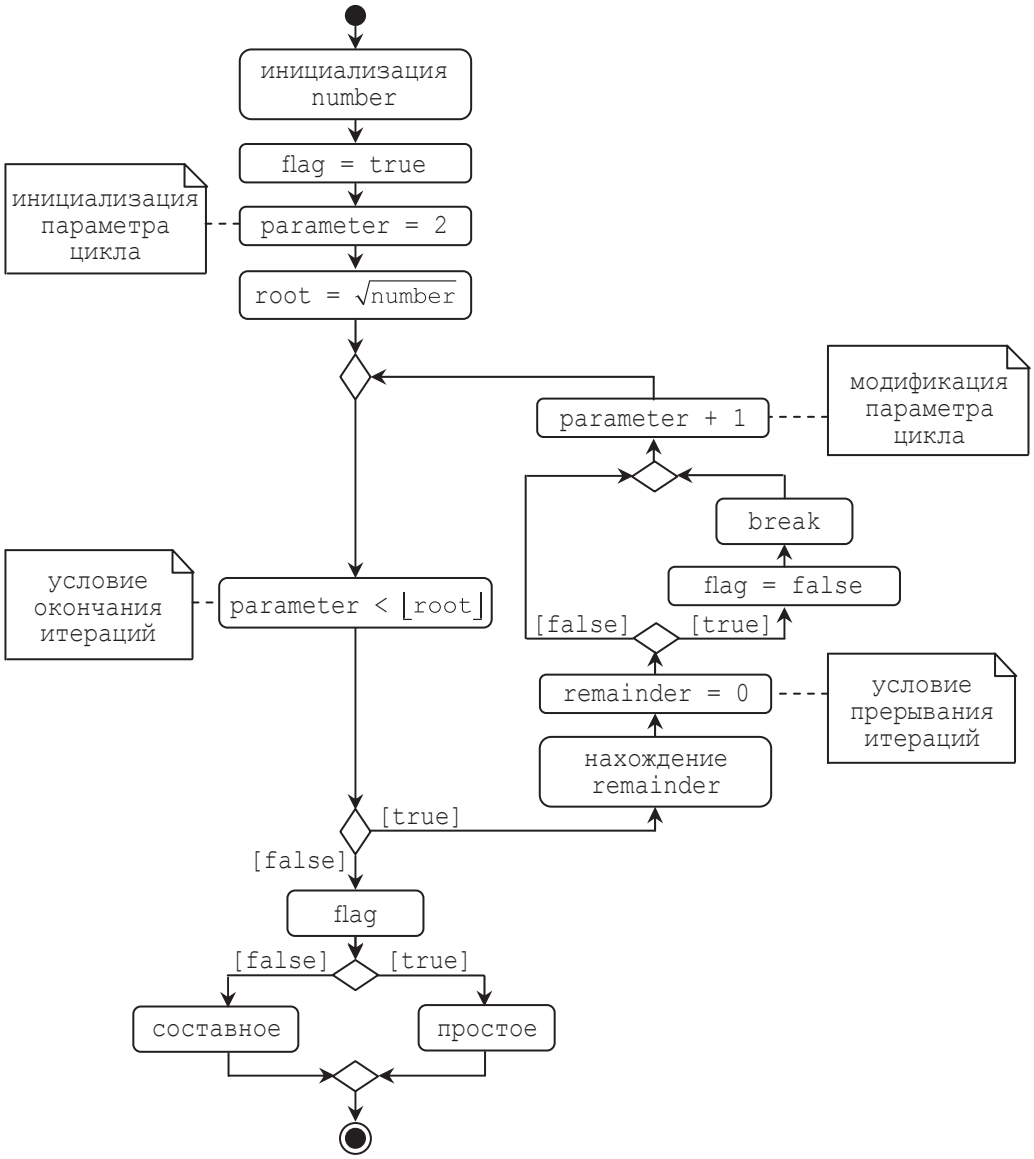



**Рис. 7.27.** Модель алгоритма тестирования простоты натурального числа методом полного перебора возможных делителей

В начальной части модели выполняется действие по инициализации тестируемого натурального числа `number`. Затем — действие по инициализации вспомогательной переменной с именем `flag` (флаг). Начальное значение этой переменной равно `true`. Значение `flag` изменяется на `false` в том случае, когда среди перебираемых натуральных чисел обнаруживается делитель тестируемого числа. Это является индикатором того, что тестируемое число является составным. После начальной инициализации переменной `flag` выполняется действие, по нахождению квадратного корня тестируемого числа для получения наибольшего значения перебираемых натуральных чисел. Квадратный корень записывается в переменную с именем `root` (корень).

В центральной части модели находится типовая поведенческая структура, моделирующая цикл типа «вначале проверка условия, затем итерация». Итерации цикла осуществляют перебор натуральных чисел. Параметр этого цикла с именем `parameter` инициализируется числом 2. Итерации продолжаются до тех пор, пока значение параметра цикла меньше, чем $\lfloor$`root`$\rfloor$ (целая часть квадратного корня от значения `number`). Главной целью каждой итерации является определение того, является ли текущее значение параметра цикла делителем тестируемого числа. Для этого в теле цикла находится остаток от деления тестируемого числа на текущее значение параметра цикла и запись результата в переменную `remainder`. Итерации должны быть прекращены, как только обнаружится, что переменная `remainder` равна нулю. Проверка значения переменной `remainder` и остановка итерационного процесса выполняется типовой поведенческой структурой развилка (`true`-ветвь). `False`-ветвь этой развилки определяет продолжение итерационного процесса и осуществляет переход к блоку, модифицирующему значение параметра цикла.

Если происходит нормальный выход из цикла без прерывания итерационного процесса, то это означает, что не обнаружено ни одного делителя тестируемого числа в диапазоне от 2 до $\lfloor$`root`$\rfloor$ и, следовательно, значение вспомогательной переменной `flag` осталось равным `true`.

В заключительной части модели расположена структура развилка, которая в зависимости от значения переменной `flag` выводит одно из двух слов: «простое» или «составное».

Модель алгоритма, приведенную на рис. 7.27, гораздо легче отобразить в программный код, чем исходное словесное описание алгоритма. Во-первых, модель явно, при помощи символов перехода, показывает последовательность работы типовых поведенческих структур, входящих в ее состав. Во-вторых, модель детерминирует множество переменных, необходимых для реализации алгоритма. В-третьих, нам хорошо известно, каким образом типовые поведенческие структуры отображаются в программный код.

Хотя модель, приведенная на рис. 7.27, обладает значительно меньшей неопределенностью, чем исходное словесное описание, она все же не настолько определенна, чтобы можно было говорить о ее однозначном отображении в программный код на языке программирования Java. Java обладает достаточной гибкостью, чтобы отобразить одну и ту же модель в различные варианты программного кода.

На рис. 7.28 приведен один из возможных вариантов такого отображения.

```
int number,                          // тестируемое число
    parameter,                       // параметр цикла, возможный делитель
    remainder;                       // остаток от деления
boolean flag;                        // флаг равен true, если число простое
// инициализация number
flag = true;                         // предположим, что число простое
parameter = 2;                       // инициализация параметра цикла
double root = Math.sqrt(number);
while(parameter < (int)root){        // заголовок цикла
   remainder = number%parameter;
   if(remainder == 0){
      flag = false;
      break;                         // прерывание итераций
   }
   parameter++;                      // модификация параметра цикла
}
if(flag)
   System.out.println("простое");
else
   System.out.println("составное");
```

**Рис. 7.28.** Отображение модели на рис. 7.27 в программный код

Структура кода на рис. 7.28 отражает структуру модели на рис. 7.27. Первые предложения объявляют переменные, необходимые для кодирования алгоритма: `number` (число); `parameter` (параметр); `remainder` (остаток); `flag` (флаг) и `root` (корень). Переменная `flag` инициализируется значением `true`. Это означает, что мы исходим из предположения, что тестируемое число является простым. После инициализации переменной `number` (например, при помощи ввода числа с консоли оператора) находится его квадратный корень, который записывается в переменную `root`.

Цикл организуется на основе оператора `while`. Перед заголовком цикла размещается предложение, инициализирующее параметр цикла числом 2. Условие окончания итераций в заголовке цикла имеет вид `parameter < (int)root`. Целая часть вещественной переменной `root` выделяется оператором явного преобразования типа. Первое предложение тела цикла находит остаток от деления тестируемого числа (переменная `number`) на текущее значение параметра цикла (переменная `parameter`). Остаток записывается в переменную `remainder`. Затем выясняется, не является ли текущее значение параметра делителем тестируемого числа. Это делается оператором `if` с условием `remainder == 0`.

Если текущий параметр цикла является делителем тестируемого числа (остаток равен нулю), то флагу присваивается значение `false` и осуществляется прерывание итерационного процесса при помощи оператора `break`.

Если текущий параметр цикла не является делителем тестируемого числа (остаток не равен нулю), то выполняется оставшееся предложение тела цикла `parameter++`, которое модифицирует значение параметра цикла, увеличивая его на единицу.

После циклической части программы размещен оператор `if`, который проверяет значение флага. Если флаг имеет значение `true`, то выводится слово «`простое`». В противном случае — слово «`составное`».

### 7.6.3. Алгоритмы Эвклида

Наибольшим общим делителем (НОД) двух натуральных чисел a и b называется наибольшее число, на которое числа a и b делятся без остатка. НОД двух натуральных чисел существует и однозначно определен, если хотя бы одно из чисел a или b не является нулем.

Известны два алгоритма, носящие имя древнегреческого математика Эвклида, для нахождения НОД. Первый алгоритм называется «нахождение НОД вычитанием», а второй — «нахождение НОД делением». Хоть алгоритмы и носят имя Эвклида, некоторые исследования подозревают, что Эвклид не является их автором.

Алгоритм нахождения НОД вычитанием словесно можно описать следующим образом.

Шаг 1. Определяем большее число.

Шаг 2. Вычитаем из большего числа меньшее число.

Шаг 3. Если разность равна нулю, то числа равны между собой и являются НОД. Работа алгоритма завершается.

Шаг 4. Если разность не равна нулю, то большее число заменяется результатом вычитания и происходит возврат к шагу 1.

Таким образом, в основе алгоритма нахождения НОД вычитанием лежит итерационный процесс. На каждой итерации выполняются следующие действия: сравнение чисел и выбор большего; вычитание из большего числа меньшего числа; замена большего числа на результат вычитания. Итерационный процесс продолжается до тех пор, пока числа не станут равными друг другу.

Рассмотрим работу алгоритма нахождения НОД вычитанием на примере нахождении НОД для чисел 30 и 18.

Итерация 1. Из чисел 30 и 18 большим является 30. 30–18 = 12. Заменяем 30 на 12.

Итерация 2. Из чисел 12 и 18 большим является 18. 18–12 = 6. Заменяем 18 на 6.

Итерация 3. Из чисел 12 и 6 большим является 12. 12–6 = 6. Заменяем 12 на 6.

Итерация 4. Числа равны. 6–6 = 0. Разность равна 0. НОД = 6. Завершение итераций.

Алгоритм нахождения НОД делением описывается следующей последовательностью шагов.

Шаг 1. Определение большего числа.

Шаг 2. Деление большего числа на меньшее число.

Шаг 3. Если большее число делится на меньшее без остатка (меньшее число является делителем большего числа), то меньшее число является НОД. Работа алгоритма завершается.

Шаг 4. Если имеется остаток от деления, не равный нулю, то большее число заменяется остатком и происходит возврат к шагу 1.

Видно, что в основе этого алгоритма также лежит итерационный процесс, в котором на каждой итерации выполняются следующие действия: сравнение чисел и выбор большего; деление большего числа на меньшее число; замена большего числа на остаток от деления. Итерации продолжаются до тех пор, пока остаток не станет равным нулю.

Проиллюстрируем работу алгоритма НОД делением тем же примером нахождения НОД для чисел 30 и 18.

Итерация 1. Из чисел 30 и 18 большим является число 30. 30 : 18 = 1, остаток = 12. Заменяем 30 на 12.

Итерация 2. Из чисел 12 и 18 большим является число 18. 18 :12 = 1, остаток = 6. Заменяем 18 на 6.

Итерация 3. Из чисел 12 и 6 большим является число 12. 12 : 6 = 2, остаток = 0. НОД = 6. Завершение итераций.

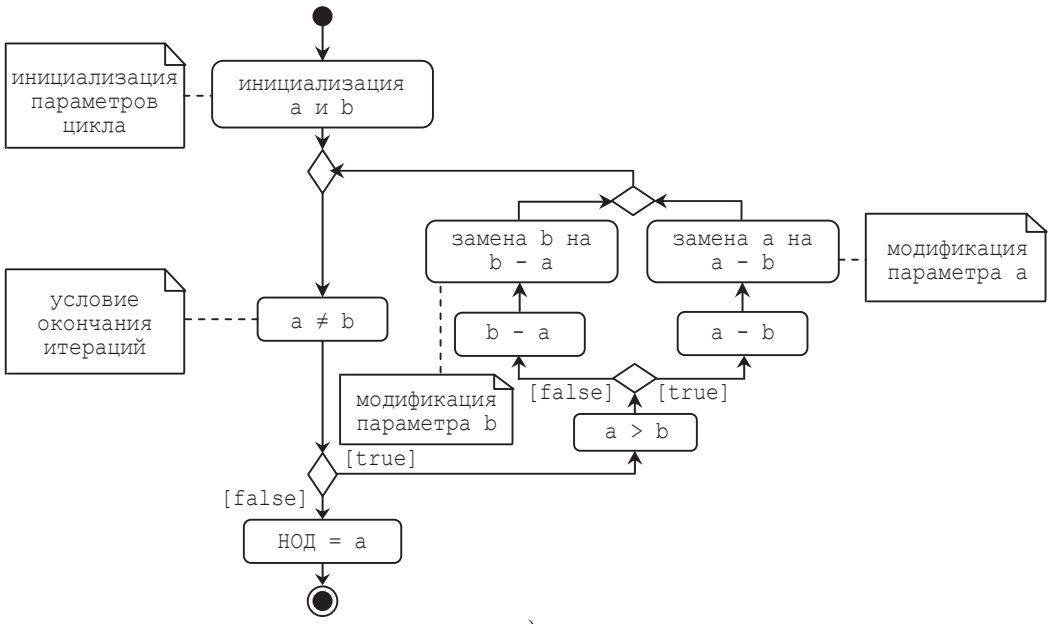
На рис. 7.29 приведены модели алгоритмов Эвклида. Основой модели на рис. 7.29 а) является структура цикл типа «вначале проверка условия, затем итерация», в итерируемый блок которой вложена структура типа развилка с двумя блоками. Параметрами цикла являются числа a и b, НОД которых определяется при помощи алгоритма. Перед вхождением в цикл его параметры инициализируются не нулевыми значениями. Условие окончания итераций записано в виде выражения $a \neq b$. Если это выражение истинно, то итерации продолжаются. Если выражение ложно, то итерации завершаются.

Итерируемый блок моделируется структурой типа развилка. Условие этой структуры, записанное в виде выражения $a > b$, позволяет определить большее из чисел a или b.

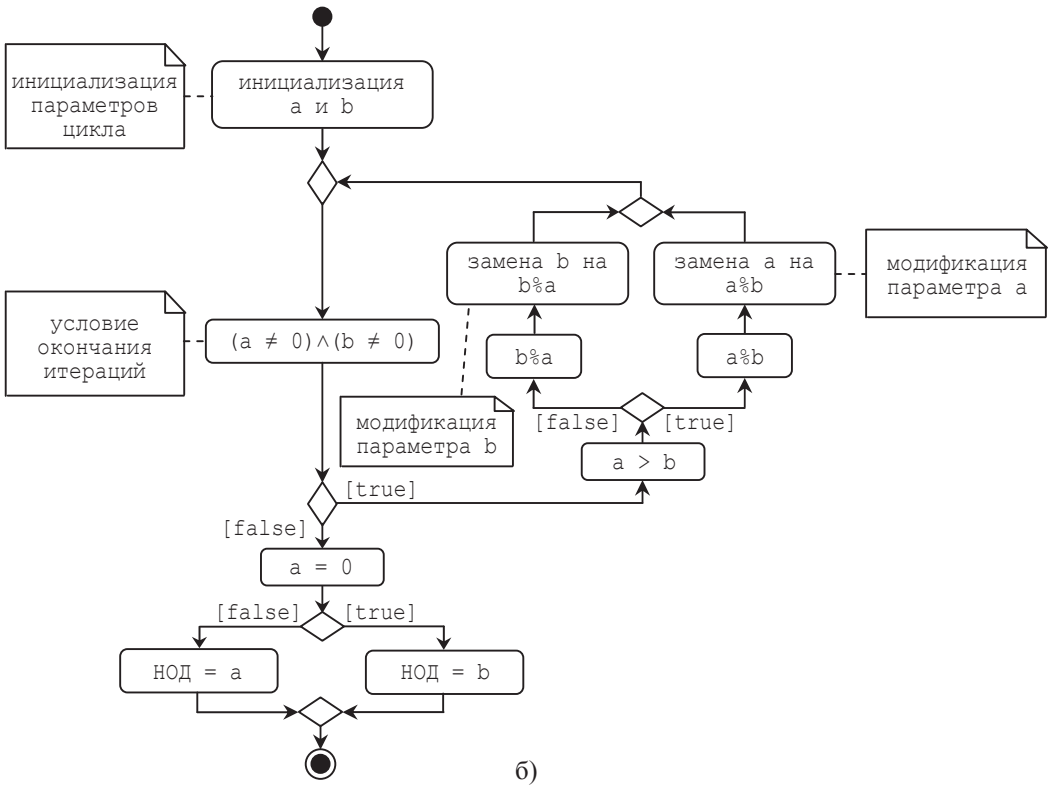
Если большим числом является число a, то выполняется true-ветвь, а если большим числом является число b, то выполняется false-ветвь структуры развилка. В каждой из этих ветвей последовательно выполняются два действия. Вначале из большего числа вычитается меньшее, а затем — большее из чисел заменяется полученной разностью.

После завершения итерационного процесса оба параметра a и b равны между собой и любой из них является искомым НОД. Завершающая часть модели представляет собой структуру типа блок, которая выводится НОД в виде значения a.

В основе модели, приведенной на рис. 7.29 б), также лежит структура цикл типа «вначале проверка, затем итерация». В итерируемый блок этой структуры вложена




а)




б)

**Рис. 7.29.** Модели алгоритмов Эвклида:
а) алгоритм нахождения НОД вычитанием; б) алгоритм нахождения НОД делением

структура типа развилка с двумя блоками. Завершающая часть модели содержит еще одну структуру типа развилка с двумя блоками. Параметрами цикла являются числа a и b, НОД которых определяется при помощи алгоритма. Перед вхождением в цикл его параметры инициализируются не нулевыми значениями.

Условие окончания итераций записано в виде выражения $(a \neq 0) \wedge (b \neq 0)$. Это выражение истинно только в том случае, когда оба числа a и b отличны от нуля. Если в итерационном процессе любое из чисел a или b станет равным нулю, то итерации прекращаются.

Для моделирования итерируемого блока используется структура типа развилка. Условие выбора одной из ветвей этой структуры записано в виде выражения $a > b$. Если это выражение истинно (число a больше числа b), то выполняется true-ветвь структуры развилка, а если — ложно (число b больше числа a), то выполняется false-ветвь. В каждой из ветвей последовательно выполняются два действия. Вначале определяется остаток от деления большего числа на меньшее, а затем — большее из чисел заменяется полученным остатком. Таким образом, в ходе итерационного процесса в параметры цикла последовательно помещаются остатки от деления большего числа на меньшее число.

Завершение итерационного процесса и выход из цикла означает, что один из параметров цикла a или b равен нулю, и, следовательно, второй из параметров является искомым НОД. Завершающая часть модели необходима для определения того, какой из параметров после завершения итераций стал равен нулю. Для этой цели используется структура типа развилка с условием выбора ветви в виде выражения $a = 0$. Если это выражение истинно, то число b является искомым НОД, а если ложно, то искомым НОД является число a.

На рис. 7.30 приведены коды моделей алгоритмов Эвклида, изображенных на рис. 7.29. С точки зрения их структурной организации, коды, приведенные на рис. 7.30 а) и 7.30 б) подобны друг другу, поскольку их модели имеют подобную структуру.

В своей начальной части оба кода содержат предложения, объявляющие и инициализирующие целочисленные переменные `a` и `b`. Эти переменные, с точки зрения пользователя кода, являются натуральными числами, для которых необходимо найти НОД, а с точки зрения программиста — параметрами цикла.

В обоих кодах структура цикл типа «вначале проверка условия, затем итерация» реализуется на основе оператора `while`. Параметрами этих циклов в обоих случаях являются переменные `a` и `b`. Отличие в организации циклов заключается в условиях окончания итераций и в выражениях, осуществляющих модификацию параметров.

В коде на рис. 7.30 а) условие окончания итераций записано в виде булева выражения `a != b`, а модификация параметров осуществляется путем вычитания из значения большего параметра значения меньшего параметра с последующей заменой большего параметра на результат вычитания. Таким образом, в цикле на рис. 7.30 а) перед каждой новой итерацией проверяется равенство параметров цикла. Итерационный процесс продолжается до тех пор, пока параметры не равны друг другу.

В коде на рис. 7.30 б) условие окончания итераций записано в виде булева выражения `(a != 0)&&(b != 0)`, а модификация осуществляется путем вычисления остатка от деления значения большего параметра на значение меньшего параметра с последующей заменой большего параметра на полученный остаток. В цикле на рис. 7.30 б) перед каждой новой итерацией проверяется, не равен ли один из параметров цикла нулю. Итерационный процесс продолжается до тех пор, пока оба параметра не равны нулю.

```
int a, b;                              // ищем НОД для a и b
// инициализация a и b
while(a != b){                         // заголовок цикла
   if(a > b)
      a -= b;
   else
      b -=a;
}
System.out.println("НОД(" + a + ", " + b + ") = " + a);
```

а)

```
int a, b;                              // ищем НОД для a и b
// инициализация a и b
while((a != 0)&&(b != 0)){             // заголовок цикла
   if(a > b)
      a %= b;
   else
      b %= a;
}
if(a=0)
   System.out.println("НОД(" + a + ", " + b + ") = " + b);
else
   System.out.println("НОД(" + a + ", " + b + ") = " + a);
```

б)

**Рис. 7.30.** Отображение моделей алгоритмов Эвклида в программный код:
а) код алгоритма нахождения НОД вычитанием; б) код алгоритма нахождения НОД делением

Если переменные `a` и `b` проинициализированы значениями 30 и 18, то после завершения работы любого из кодов, приведенных на рис. 7.30, на экран будет выведена следующая строка

```
НОД(30, 18) = 6
```

### 7.6.4. Вычисление факториала

В математике изучаются последовательности чисел, в которых каждый следующий член последовательности выражается через предыдущие. Формула, позволяющая выразить каждый из членов последовательности через один или несколько предыдущих членов, называется *рекуррентной формулой*, а последовательность, члены которой удовлетворяют некоторой рекуррентной формуле — *рекуррентной последовательностью*. Примерами рекуррентной последовательности являются арифметическая и геометрическая прогрессии. В арифметической прогрессии, например, каждый следующий член равен предыдущему, увеличенному на постоянную величину, называемую разностью прогрессии. Рекуррентная формула для арифметической прогрессии записывается в виде

$$a_n = a_{n-1} + d$$

В основе большого количества итерационных математических алгоритмов лежат рекуррентные формулы и рекуррентные последовательности. В последующих подразделах рассмотрены примеры кодирования математических алгоритмов, основанных на рекуррентных формулах и последовательностях. В качестве первого примера этого класса алгоритмов рассмотрим алгоритм вычисления факториала натурального числа.

*Факториалом* натурального числа *n* называется произведение всех натуральных чисел от единицы до *n* включительно

$$n! = 1 \cdot 2 \cdot \ldots \cdot n = \prod_{i=1}^{n} i$$

Факториал натурального числа *n* можно рассматривать как значение *n*-го элемента следующей рекуррентной последовательности

$$1! = 0! \cdot 1, \quad 2! = 1! \cdot 2, \quad 3! = 2! \cdot 3, \ \ldots, \quad n! = (n-1)! \cdot n$$

Принято, что факториал нуля равен единице. Рекуррентная формула, позволяющая выразить *n*-й член рекуррентной последовательности факториалов, имеет вид

$$n! = \begin{cases} 1, & n = 0 \\ n \cdot (n-1)!, & n > 0 \end{cases}$$

Алгоритм, вычисляющий факториал для заданного натурального числа, весьма полезен и применим во многих случаях. В комбинаторике, например, факториалом натурального числа *n* определяется количество перестановок элементов множества, состоящего из *n* элементов. Если имеется множество {A, B, C}, то существует 3! = 6 перестановок элементов этого множества: ABC, ACB, BAC, BCA, CAB, CBA.

Понятия натуральное число в математике и целочисленное данное в программировании не тождественны. В математике под натуральным числом мы можем понимать сколь угодно большое число. Иными словами, количество разрядов натурального числа, которым мы оперируем в математических рассуждениях, не ограничено. В языке программирования Java существуют типы для представления целых чисел. Каждый тип предполагает фиксированное и, следовательно, ограниченное количество бит для представления числа. Наибольшее натуральное число, которым может оперировать программист, имеет тип `long`, который предоставляет 64 бита для записи целого числа. Несложно посчитать, что наибольшее положительное число, которое можно записать в переменную типа `long`, в десятичной системе счисления равно

9 223 372 036 854 775 807

Таким образом, если для представления целого числа использовать переменную типа `long`, то это число не должно иметь более 19 десятичных разрядов. Факториал числа 20 представляется 19-разрядным десятичным числом и равен

20! = 2 432 902 008 176 640 000

Факториал числа 21 представляется уже 20-разрядным числом. Следовательно, в алгоритме для вычисления факториала целесообразно контролировать величину натурального числа и начинать вычисление факториала в том случае, когда это число не больше, чем число 20.

На рис. 7.31 приведена модель алгоритма вычисления факториала натурального числа `number`. В начальной части алгоритма расположена структура типа блок, ответственная за инициализацию натурального числа. Остальная часть алгоритма построена на основе структуры развилка с двумя блоками. В блок `true`-ветви структуры развилка вложена структура цикл типа «вначале проверка условия, затем итерация». Условие структуры развилка записано в виде выражения `number < 20`, которое проверяет, не превышает ли величина `number` числа 20. Если `number` больше 20, то выводится строка «число больше 20», и алгоритм завершает работу. Если `number` меньше 20, то начинается вычисление его факториала.

Цикл типа «вначале проверка условия, затем итерация» осуществляет вычисление факториала. В основе вычислений лежит рекуррентный итерационный процесс. На каждой итерации вычисляется очередной член рекуррентной последовательности в соответствии с рекуррентной формулой. Рекуррентность алгоритма проявляется в том, что вычисления в каждой текущей итерации используют результат, полученный в предыдущей итерации.

Перед вхождением в цикл производится инициализация переменных `factorial` и `i`. Переменная `factorial` в процессе итераций последовательно принимает значения рекуррентной последовательности факториалов, а после завершения итерационного процесса содержит искомое значение факториала. Переменная `i` является параметром цикла.

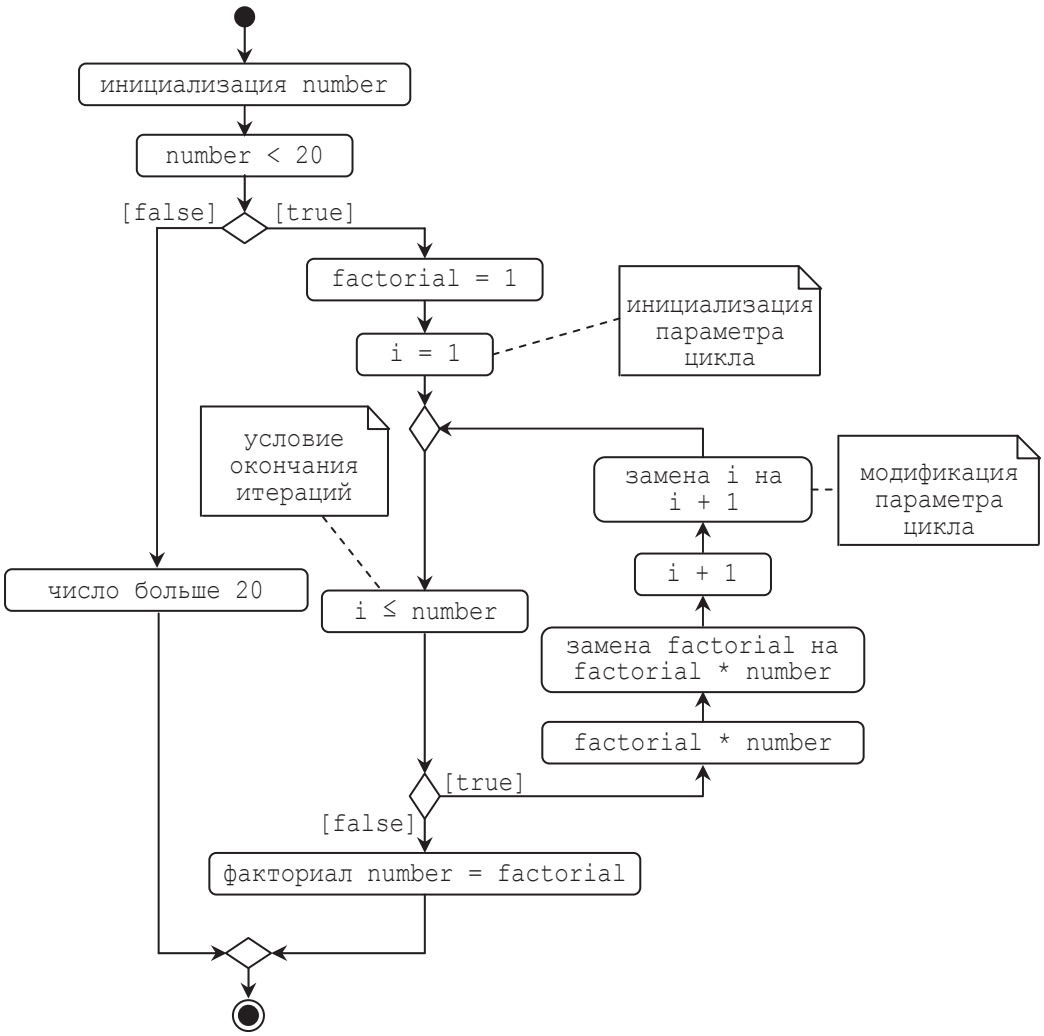



**Рис. 7.31.** Модель алгоритма вычисления факториала

Если факториал найден, то в завершающей части алгоритма выводится строка со значением найденного факториала.

На рис. 7.32 приведено отображение модели алгоритма вычисления факториала в программный код.

Код на рис. 7.32 является относительно простым и понятным. Отметим его особенности. Целочисленная переменная с именем `factorial`, которая в процессе работы кода последовательно принимает значения факториалов чисел от 0 до значения переменной `number`, имеет тип `long`, позволяющий хранить наибольшее возможное натуральное число среди всех допустимых целочисленных типов языка программирования Java. Цикл организован на основе оператора `for`, поскольку до вхождения в цикл нам заранее известно количество итераций.

```
int number;                          // число, для которого ищется факториал
long factorial = 1;                  // начальное значение факториала 1
// инициализация number
if(number < 20){
   for(int i = 1; i <= number; i++) // заголовок цикла
      factorial *= number;
      System.out.println("факториал " + number + "= " + factorial);
   }
else
   System.out.println("число больше 20");
```

**Рис. 7.32.** Отображение модели вычисления факториала в программный код

### 7.6.5. Вычисление чисел Фибоначчи

Известна рекуррентная последовательность, изобретенная итальянским математиком Леонардо Фибоначчи и состоящая из натуральных чисел, названных его именем. *Последовательность чисел Фибоначчи* имеет вид.

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

Нетрудно заметить, что в приведенной последовательности чисел любое число $f_i$, начиная с третьего, равно сумме двух предыдущих чисел этой последовательности. Числа Фибоначчи обладают рядом интересных свойств. Например, соседние числа Фибоначчи являются взаимно простыми; НОД двух чисел Фибоначчи является тоже число Фибоначчи; число Фибоначчи четное тогда и только тогда, когда его номер кратный трем и др. Рекуррентная формула для последовательности чисел Фибоначчи может быть записана в виде

$$f_i = \begin{cases} 1, & i = 1 \\ 1, & i = 2 \\ f_{i-1} + f_{i-1}, & i > 2 \end{cases}$$

Полезной является программа, которая вычисляет значение числа Фибоначчи по заданному индексу последовательности.

Числа в последовательности Фибоначчи быстро возрастают, и очевидно, что, начиная с некоторого индекса, они становятся больше, чем наибольшее число, которое может располагаться в переменной типа `long`. Поэтому, прежде чем конструировать алгоритм вычисления числа Фибоначчи по заданному индексу, целесообразно определить предельный индекс, при котором программа работает корректно. Поскольку наибольшее число, которое может быть помещено в переменную типа

`long` является 19-разрядным, то нам необходимо определить значение индекса такого числа Фибоначчи, после которого все последующие числа Фибоначчи представляются более чем 19 разрядами. Таким предельным индексом является 90, которому соответствует следующее число Фибоначчи

2 880 067 194 370 816 000

Построим модель алгоритма вычисления числа Фибоначчи по его индексу, который, как только что выяснилось, может изменяться в пределах от 1 до 90. На рис. 7.33 изображена искомая модель в виде диаграммы деятельности.

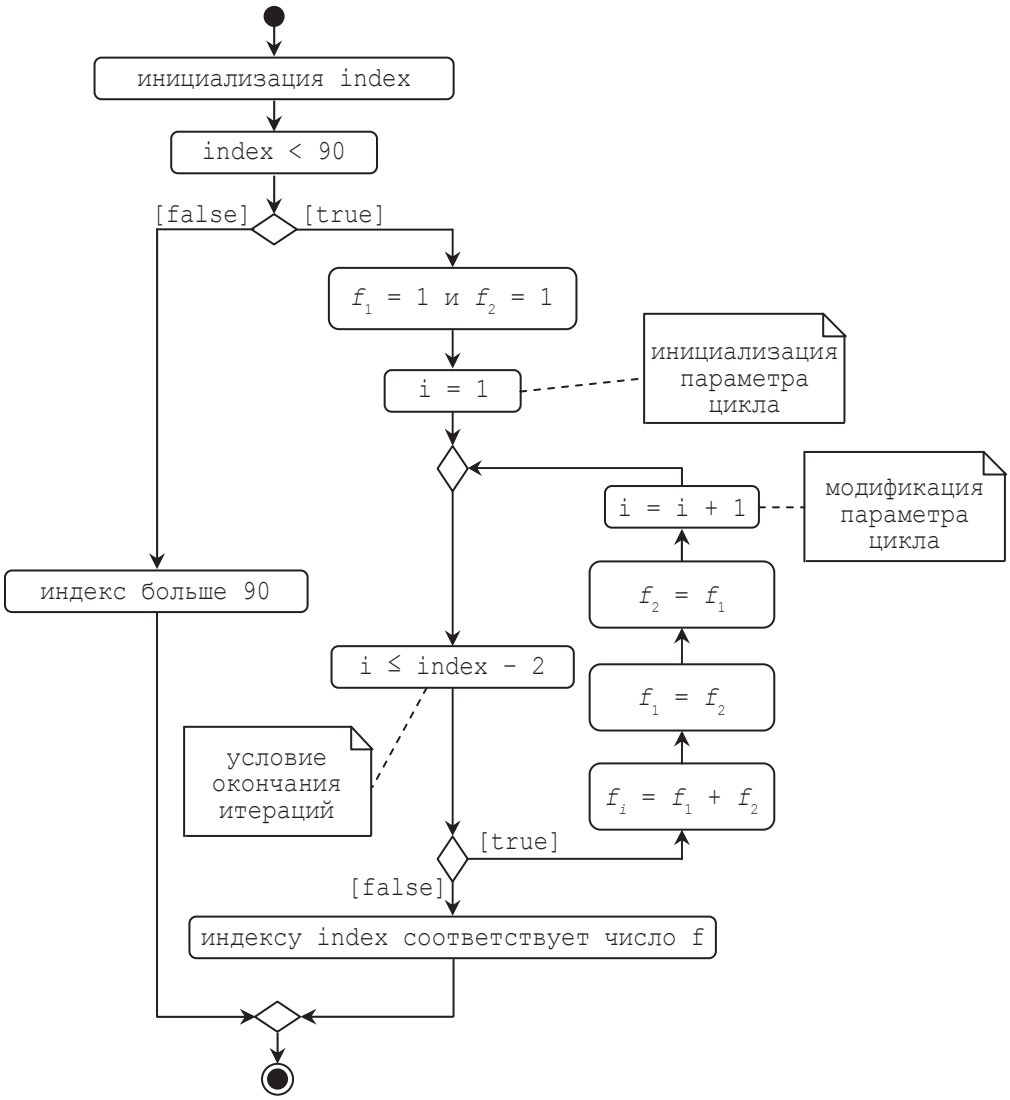



**Рис. 7.33.** Модель алгоритма вычисления чисел Фибоначчи

Модель алгоритма вычисления числа Фибоначчи по его индексу в последовательности, с точки зрения ее структуры, подобна модели алгоритма вычисления факториала. В начальной части алгоритма расположена структура типа блок, осуществляющая инициализацию индекса. Основная часть алгоритма построена на основе структуры развилка с двумя блоками. В блок `true`-ветви структуры развилка вложена структура цикл типа «вначале проверка условия, затем итерация». Условие структуры развилка записано в виде выражения `index < 90`, которое проверяет, не превышает ли индекс числа 90. Если индекс больше 90, то выводится строка «индекс больше 90», и алгоритм завершает работу. Если индекс меньше 90, то начинается вычисление числа Фибоначчи при помощи структуры цикл типа «вначале проверка условия, затем итерация».

Цикл реализует рекуррентный итерационный процесс. На каждой итерации вычисляется очередное число в соответствии с рекуррентной формулой последовательности Фибоначчи. Это выполняется следующим образом. Перед вхождением в цикл инициализируются две переменные $f_1$ и $f_2$. Для некоторого числа $f_i$ переменные $f_1$ и $f_2$ содержат числа, которые ему предшествуют. На каждой итерации вначале вычисляется очередной член последовательности по формуле $f_i = f_1 + f_2$, а затем переменные $f_1$ и $f_2$ подготавливаются к следующей итерации.

Для заданного индекса количество итераций должно быть на 2 меньше, чем значение индекса. Например, для вычисления шестого числа Фибоначчи необходимо выполнить четыре итерации. Читателю предлагается экспериментально проверить справедливость этого утверждения. Поэтому условие окончания итераций записано в виде `i ≤ index − 2`. После завершения итерационного процесса переменная $f_i$ принимает значение искомого числа.

Если число Фибоначчи вычислено, то в завершающей части алгоритма выводится строка со значениями индекса и соответствующего ему числа Фибоначчи.

```
int index;                                    // индекс
// инициализация index
if(index < 90) {
   long f;                                    // текущее число
   long f1 = 1;                               // 1-е предшествующее число
   long f2 = 1;                               // 2-е предшествующее число
   for(int i = 1; i <= index - 2; i++) {      // заголовок цикла
      f = f1 + f2;
      f1 = f2;
      f2 = f;
   }
   System.out.println("индексу " + index + (" соответствует число " + f));
}
else
   System.out.println("индекс больше 90");
```

**Рис. 7.34.** Отображение модели вычисления числа Фибоначчи в программный код

На рис. 7.34 приведен код, в который можно отобразить модель алгоритма вычисления числа Фибоначчи. Отметим его особенности. Переменные `f`, `f1` и `f2`, которые соответствуют текущему и двум предшествующим числам последовательности Фибоначчи, имеют тип `long` для того, чтобы можно было оперировать как можно большими числами. Поскольку количество итераций цикла известно заранее, то целесообразно цикл организовать на основе оператора `for`.

### 7.6.6. Вычисление полинома по схеме Горнера

Полиномы образуют универсальный класс функций. Их универсальность заключается в способности полинома аппроксимировать другие функции. Полином n-ой степени представим в виде.

$$p_n(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} \ldots + a_n$$

Схемой Горнера называют алгоритм вычисления значения полинома n-ой степени для заданного аргумента полинома x. Алгоритм назван в честь английского математика Уильяма Горнера, который опубликовал этот алгоритм в начале 19 века.

Рассмотрим последовательность полиномов.

$$p_0(x), p_1(x), p_2(x), p_3(x), \ldots$$
$$p_0(x) = a_0;$$
$$p_1(x) = a_0x + a_1;$$
$$p_2(x) = a_0x^2 + a_1x + a_2;$$
$$p_3(x) = a_0x^3 + a_1x^2 + a_2x + a_3;$$
$$\ldots$$

Приведенная последовательность полиномов является рекуррентной, поскольку каждый последующий член последовательности можно выразить через предыдущий.

$$p_1(x) = p_0(x) * x + a_1;$$
$$p_2(x) = p_1(x) * x + a_2;$$
$$p_3(x) = p_2(x) * x + a_3;$$
$$\ldots$$

Рекуррентная формула для приведенной последовательности полиномов имеет вид

$$p_n(x) = p_{n-1}(x) * x + a_n;$$

Таким образом, вычисление значения полинома *n*-ой степени для заданного *x* можно выполнить при помощи итерационного процесса, в основе которого лежит рекуррентная формула. Итерационный процесс имеет два параметра: (1) переменная степень полинома, изменяющаяся от 0 до *n* и (2) переменный коэффициент, изменяющийся от $a_0$ до $a_n$.

На рис. 7.35 приведена модель алгоритма, построенного на основе схемы Горнера. Для заданных *n* и *x* алгоритм вычисляет значение полинома *n*-ой степени при следующих значениях коэффициентов $a_0=1$, $a_1=2$, $a_2=3$ и т. д.

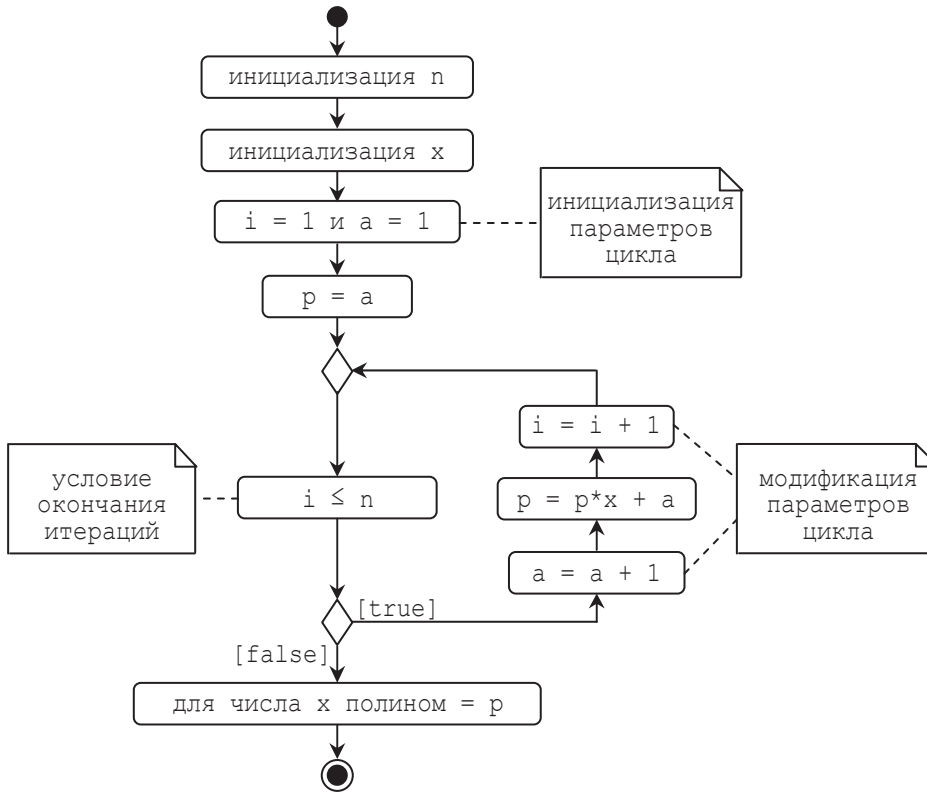



**Рис. 7.35.** Модель алгоритма Горнера

Для упрощения алгоритма принято, что коэффициенты принимают значения $a_0=1$, $a_1=2$, $a_2=3$ и т. д., и, следовательно, могут вычисляться в ходе итерационного процесса как один из его параметров. Таким образом, алгоритм вычисляет значение полинома вида.

$$p_n(x) = x^n + 2x^{n-1} + 3x^{n-2} \ldots$$

Ясно, что в общем случае значения коэффициентов могут изменяться по более изощренному правилу.

Начальная часть алгоритма состоит из двух структур типа блок, которые осуществляют инициализацию степени и аргумента полинома (переменные n и x соответственно).

Центральной частью алгоритма является цикл типа «вначале проверка условия, затем итерация». Параметрами цикла являются: степень полинома (переменная i) и коэффициент (переменная a). Перед входом в цикл формируется значение полинома нулевой степени при помощи выражения p = a. Поэтому количество итераций, которые выполняет цикл, на единицу меньше количества членов полинома и равно его степени. На каждой итерации выполняются действия по вычислению значение полинома i-ой степени, начиная с первой, а также действия по модификации параметров цикла. Эти действия выполняются в следующей последовательности. Вначале модифицируется параметр a, что эквивалентно заданию очередного коэффициента. Затем вычисляется значение полинома i-ой степени (начиная с первой) при помощи предложения p = p*x + a, соответствующего рекуррентной формуле алгоритма Горнера. Затем увеличивается на единицу значение параметра i, что эквивалентно увеличению степени полинома на единицу.

В заключительной части алгоритма располагается структура типа блок, при помощи которой выводится найденное значение полинома.

Недостатком алгоритма является отсутствие ограничения на значение переменной n, задающей степень полинома. Такое ограничение необходимо, поскольку ясно, что значения полинома высоких степеней могут превосходить возможности компьютерного представления числовых данных.

На рис. 7.36 приведено отображение модели алгоритма Горнера в программный код.

```
int n;                                    // степень полинома
int a = 1;                                // коэффициент полинома
double x;                                 // аргумент полинома
double p = a;                             // значение полинома
// инициализация n и x
   for(int i = 1; i <= n + 1; i++) {      // заголовок цикла
      a++;
      p = p * x + a;
   }
   System.out.println("для числа " + x + " полином = " + p);
```

**Рис. 7.36.** Отображение модели алгоритма Горнера в программный код

Код, приведенный на рис. 7.36, аналогичен ранее приведенным примерам кодов итерационных процессов, поэтому нет необходимости подробно его комментировать. Отметим лишь, что для реализации цикла типа «в начале проверка условия, затем итерация» использован оператор for, поскольку количество итераций заранее известно.

### 7.6.7. Вычисление экспоненциальной функции при помощи ряда Тейлора

Рядом Тейлора называют представление некоторой функции в виде бесконечной суммы степенных функций. Ряд назван в честь английского математика Брука Тейлора. Например, экспоненциальную функцию $y=e^x$ можно представить в виде ряда Тейлора следующим образом:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Приведенный ряд Тейлора позволяет вычислить приближенное значение экспоненциальной функции с любой, наперед заданной, точностью. Точность зависит от количества слагаемых ряда Тейлора, использованных при вычислении. Чем больше слагаемых, тем выше точность.

Запишем ряд Тейлора для вычисления экспоненциальной функции в виде.

$$e^x = t_0(x) + t_1(x) + t_2(x) + \ldots + t_{n-1}(x) + t_n(x) + \ldots$$

Члены рассматриваемого ряда Тейлора находятся в рекуррентном соотношении, и каждый последующий выражается через предыдущий.

$$t_1(x) = t_0(x) * \frac{x}{1};$$

$$t_2(x) = t_1(x) * \frac{x}{2};$$

$$t_3(x) = t_2(x) * \frac{x}{3};$$

$$\ldots$$

$$t_n(x) = t_{n-1}(x) * \frac{x}{n}$$

Рекуррентная формула имеет вид

$$t_n(x) = t_{n-1}(x) * \frac{x}{n}$$

Вычисление экспоненциальной функции при помощи ряда Тейлора производится при помощи итерационного процесса. На каждой итерации вычисляется очередной член рекуррентной последовательности ряда Тейлора, который затем прибавляется к сумме предыдущих членов. Итерационный процесс продолжается до тех пор, пока абсолютная величина разности результатов двух следующих друг за другом итераций не станет меньше, чем наперед заданное число $\varepsilon > 0$. Поскольку результаты двух следующих друг за другом итераций отличаются на величину

последнего вычисленного члена рекуррентной последовательности, то условие завершения итерационного процесса представимо в виде

$$|t_n(x)| < \varepsilon$$

Словами это условие формулируется следующим образом: «Итерационный процесс завершается, когда очередной член ряда Тейлора, по модулю, становится меньше, чем ε».

Приведенного описания способа вычисления экспоненциальной функции с наперед заданной точностью на основе ряда Тейлора достаточно для синтеза модели алгоритма, которая приведена на рис. 7.37.

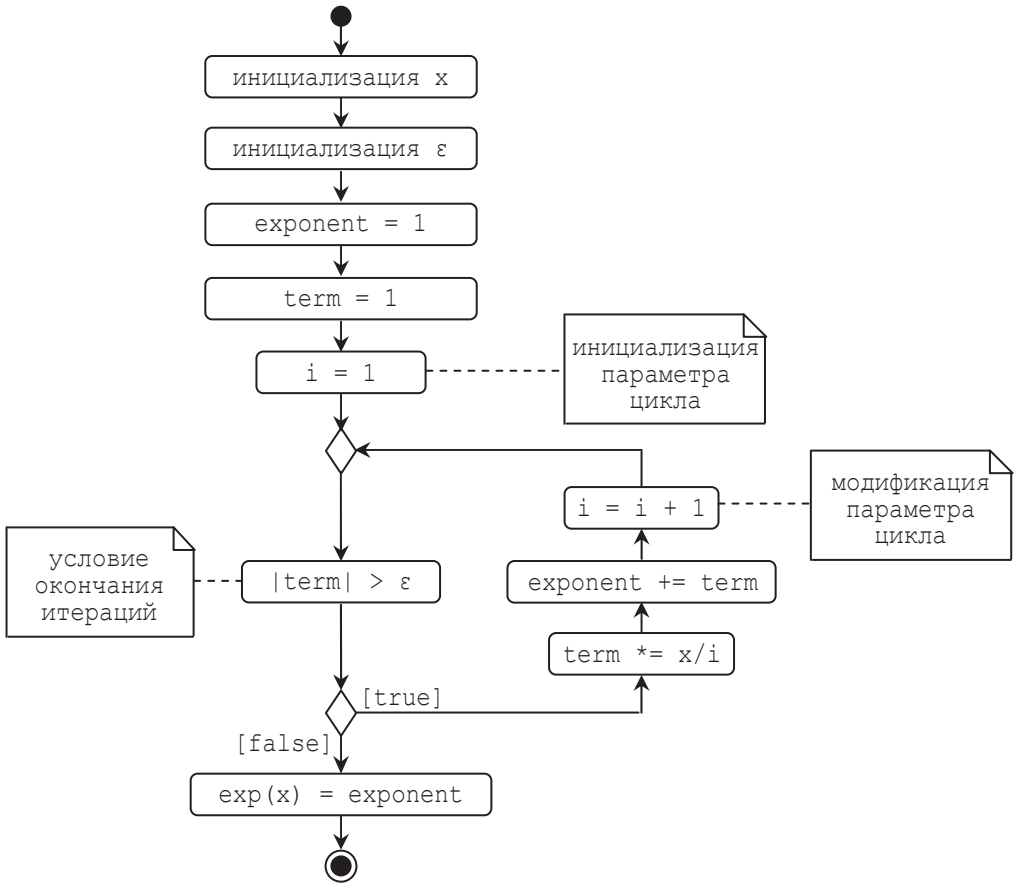



**Рис. 7.37.** Модель алгоритма вычисления экспоненциальной функции на основе ряда Тейлора

В начальной части алгоритма размещены пять структур типа блок, при помощи которых последовательно выполняют следующие действия: (1) инициализируется значение аргумента экспоненциальной функции *x*; (2) инициализируется заданная

точность вычислений ε; (3) текущее значение экспоненциальной функции `exponent` инициализируется единицей; (4) текущий член ряда Тейлора `term` инициализируется единицей; (4) инициализируется параметр цикла `i`. Перед началом итераций в переменную `exponent` записывается значение нулевого члена ряда Тейлора. Итерации начинаются с вычисления первого члена ряда Тейлора. После завершения итерационного процесса переменная `exponent` содержит искомое значение экспоненциальной функции

Центральной частью алгоритма является цикл типа «вначале проверка условия, затем итерация». Цикл имеет один параметр `i`, который в ходе итерационного процесса принимает значения, соответствующие номерам членов ряда Тейлора. Цикл продолжается до тех пор, пока абсолютная величина текущего члена ряда Тейлора `term` больше, чем заданная точность вычислений ε. На каждой итерации выполняются действия по вычислению очередного члена ряда Тейлора и прибавления его к текущему значению экспоненциальной функции `exponent`, а также действия по модификации параметров цикла.

В заключительной части алгоритма располагается структура типа блок, при помощи которой выводится найденное значение экспоненциальной функции.

На рис. 7.38 приведено отображение модели алгоритма вычисления экспоненциальной функции в программный код.

```
float x;                                        // аргумент экспоненты
double precisn,                                 // точность вычислений
       exponent = 1,                            // значение экспоненты
       term = 1;                                // член ряда Тейлора
// инициализация x и precisn
int i = 1;                                      // параметр цикла
   while(Math.abs(term) > precisn){             // заголовок цикла
      term *= x/i;
      exponent += term;
      i++;
   }
   System.out.println("exp(" + x + ") = " + exponent);
```

**Рис. 7.38.** Отображение модели алгоритма вычисления экспоненциальной функции в программный код

Поскольку заранее не известно, какое количество итераций потребуется для достижения заданной точности вычислений, цикл организован на основе оператора `while`. В заголовке цикла записано булево выражение, определяющее условие окончания итераций в виде неравенства.

```
Math.abs(term) > precisn
```

В правой части неравенства указано имя переменной `precisn` (точность), содержащей значение заданной точности вычислений. В левой части неравенства записано сообщение, при помощи которого вызывается метод с именем `abs`, определенный в классе `Math`. Метод возвращает абсолютную величину переменной `term`, переданной ему в качестве параметра. Особенности вызова методов предопределенного класса `Math` изучаются во второй части книги. Булево выражение истинно до тех пор, пока абсолютная величина текущего члена ряда Тейлора `term` меньше, чем заданная точность вычислений `precisn`. В теле цикла тремя предложениями последовательно вычисляется очередной член ряда, предыдущее значение экспоненциальной функции увеличивается на величину очередного члена ряда и осуществляется модификация параметра цикла.

## Упражнения для самостоятельной работы

7.1. Запишите рекуррентные формулы для следующих рекуррентных последовательностей.

$$1-x^3+x^6-x^9+\ldots$$

$$1-x^2-x^4-x^6-\ldots$$

$$\frac{1}{x}+\frac{2}{x^2}+\frac{3}{x^3}+\frac{4}{x^4}+\ldots$$

$$\frac{1!}{x}+\frac{2!}{x}+\frac{3!}{x}+\frac{4!}{x}+\ldots$$

7.2. Разработайте программный код для вычисления значений приведенных ниже выражений. Перед разработкой кода запишите соответствующие рекуррентные формулы.

$$1+\frac{1}{2!}+\frac{1}{3!}+\ldots+\frac{1}{n!}$$

$$\underbrace{\sqrt{2+\sqrt{2+\ldots+\sqrt{2+\sqrt{2}}}}}_{n\ раз}$$

$$\sin x+\sin x^2+\ldots+\sin x^n$$

$$\sin(x)+\sin(\sin(x))+\ldots+\underbrace{\sin(\sin(\ldots\sin(x)))}_{n\ раз}$$

7.3. Разработайте программный код, решающий следующую задачу. Пользователю последовательно выводятся запросы на ввод произвольного числа. Запросы продолжаются до тех пор, пока каждое последующее число, которое вводит пользователь, больше предыдущего. После окончания ввода чисел пользователю сообщается, какое количество чисел было введено.

7.4. Разработайте программный код, который подсчитывает количество цифр в некотором целом положительном числе $k$.

7.5. Разработайте программный код, который находит двузначное натуральное число, равное утроенному произведению его цифр.

7.6. Разработайте программный код, который находит сумму квадратов четных чисел от 2 до 20 (2, 4, 6, 8, 10, 12, 14, 16, 18, 20).

7.7. Разработайте программный код, который находит все пятизначные натуральные числа, сумма цифр которых больше $N$, но меньше $2 \cdot N$.

7.8. Разработайте программный код, который определяет, является ли заданное число совершенным, т. е. равным сумме всех своих собственных положительных делителей, отличных от самого числа (например, число 6 является совершенным: 6 = 1 + 2 + 3).

7.9. Разработайте программный код, который вычисляет произведение m членов арифметической прогрессии, если известны значение первого члена $a_1$ и шаг арифметической прогрессии $h$.

7.10. Разработайте программный код, который определяет количество простых чисел в заданном интервале натуральных чисел от $n$ до $m$.

7.11. Разработайте программный код, который находит первое число Фибоначчи большее, чем заданное число $m$.

7.12. Разработайте программный код, вычисляющий по схеме Горнера значение следующего полинома:

$$y(x) = 11x^{10} + 10x^9 + 9x^8 + \ldots + 2x + 1$$

7.13. Лейбниц доказал, что для числа π справедливо следующее соотношение

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots$$

Разработайте программный код, который находит приближенное значение числа π путем суммирования 100 первых членов приведенного ряда.

7.14. Известно, что последовательность вида

$$1-\frac{x^2}{2!}+\frac{x^4}{4!}-\frac{x^6}{6!}+\ldots+(-1)^n\frac{x^{2n}}{(2n)!}$$

сходится к функции cos($x$). Разработайте программный код, который на основе приведенной последовательности находит приближенное значение функции cos($x$) с точностью $\varepsilon = 10^{-6}$.

Раздел 8

# МАССИВЫ. РАБОТА С МАССИВАМИ

*Массивом называется набор проиндексированных и однотипных данных, имеющих общее имя*. В приведенном определении массива отражены два отличительных свойства массива как поименованного набора данных.

Во-первых, все данные, входящие в массив (элементы массива) должны быть одного типа. Это может быть либо один из примитивных типов, либо тип одного из предопределенных классов (например, класса `String`), либо новый тип, введенный программистом и определенный им при помощи класса или интерфейса.

Во-вторых, данные, входящие в массив, должны быть проиндексированы. Слово «проиндексированы» здесь является синонимом слова «пронумерованы» и означает, что каждый элемент массива снабжен уникальным индексом-номером. Принято, что *нумерация элементов массива всегда начинается с нуля*. Поэтому, если, например, массив состоит из 10 элементов, последовательно расположенных друг за другом, то индекс первого элемента будет равен 0, а индекс последнего — 9. Поскольку значения индексов уникальны, то индекс удобно использовать как средство идентификации элементов массива. Для того, чтобы получить прямой доступ к некоторому элементу массива, достаточно указать имя массива и значение индекса его элемента.

Каждый элемент массива может быть проиндексирован одним, двумя, тремя или, в общем случае, *n* индексами. Если элемент массива снабжен только одним индексом, то такой массив называется одномерным. Если элемент массива снабжен двумя индексами, то массив называется двухмерным. Ясно, что для случая трех индексов получаем трехмерный массив, а в общем случае — *n*-мерный массив. При решении задач императивного программирования иногда полезно одномерный массив ассоциировать с вектором, а двухмерный — с плоской таблицей или матрицей.

Поскольку индексы — это номера, то, с точки зрения программного кода, индексы представляются целочисленными данными одного из примитивных типов. В практике императивного программирования переменные, хранящие значения индексов массива, часто являются параметрами простого или вложенных циклов. Это позволяет на каждой итерации обеспечивать доступ к очередному элементу массива.

## 8.1. Объявление, создание и начальная инициализация одномерного массива

В языке программирования Java рассматриваются не отдельные массивы, а класс массивов. Класс массивов представляет общие свойства и поведение массивов и не может использоваться для хранения данных. Массив данных, которыми можно оперировать в коде, — это *объект класса массивов*. Поэтому, прежде чем оперировать с массивом, необходимо создать соответствующий объект-массив. В процедуре создания объекта-массива можно выделить следующие действия:

- объявление имени массива и типа его элементов;
- создание массива, например, при помощи предложения со служебным словом `new`;
- инициализация элементов массива начальными значениями.

Имя массива является, по сути, именем ссылочной переменной, предназначенной для хранения ссылки на объект класса массивов.

Рассмотрим, каким образом кодируются объявление, создание и инициализация массива на примере *класса одномерных массивов*.

Объявление имени массива и типа его элементов можно выполнить при помощи отдельного предложения, имеющего следующие варианты нотации.

```
<тип элементов массива>[ ] <имя массива>;
<тип элементов массива> <имя массива>[ ];
```

Признаком того, что объявляется ссылка на массив, а не простая локальная переменная, является пара квадратных скобок. Нотации приведенных выше предложений отличаются местом расположения этих квадратных скобок. Скобки могут располагаться либо после имени типа элементов массива (верхняя строчка), либо после имени массива (нижняя строчка). На рис. 8.1 приведены примеры объявления одномерных массивов, в которых квадратные скобки располагаются после имени типа элементов массива.

```
float[] temp;              // объявлена ссылка на массив с именем temp,
                           // состоящий из элементов типа float

String[] drivers;          // объявлена ссылка на массив с именем drivers,
                           // состоящий из ссылок на объекты класса String

Telephone[] phoneNumbers; // объявлена ссылка на массив
                           // с именем phoneNumbrs, состоящий из ссылок
                           // на объекты класса Telephone
```

**Рис. 8.1.** Примеры объявления одномерного массива.
Квадратные скобки располагаются после имени типа элементов массива

Приведенный способ объявления массива удобно использовать в тех случаях, когда при помощи одного предложения нужно объявить несколько ссылок на массивы, состоящие из элементов одного и того же типа. Например,

```
double[] firstTest, secondTest, thirdTest;
```

В приведенном примере при помощи одного предложения объявляются сразу три одномерных массива с именами `firstTest`, `secondTest`, `thirdTest`, состоящих из элементов типа `double`.

Способ объявления массива, когда квадратные скобки располагаются после имени массива, удобно использовать в тех случаях, когда при помощи одного предложения объявляются и массивы, и обычные переменные одного и того же типа. Например,

```
String teacher, group[];
```

В приведенном примере объявлена обычная переменная с именем `teacher` и массив с именем `group`. И обычная переменная `teacher`, и массив `group` предназначены для хранения данных типа `String`.

С точки зрения компилятора, предложения, объявляющие массив, означают, что необходимо создать переменную ссылочного типа, которая предназначена для хранения ссылки на объявленный массив. Однако самого массива еще нет. Для того, чтобы он появился, необходимо написать предложение, при помощи которого создается объект-массив.

Если ссылочная переменная (имя массива) уже объявлена, то при создании массива необходимо только детерминировать количество его элементов. Ранее мы узнали, что в языке программирования Java новые объекты создаются при помощи предложения со служебным словом `new`. В случае создания объекта-массива это предложение имеет следующую структуру:

```
<имя массива> = new <тип элементов массива> [<кол-во элементов массива>];
```

Например, после того как объявлен массив с именем `temp`, можно его создать при помощи следующего предложения:

```
temp = new float[6];
```

Предложение декларирует, что объявленный ранее массив с именем `temp`, состоящий из элементов типа `float`, содержит шесть элементов.

Для объявления массива и его создания не обязательно использовать два отдельных предложения. Эти действия можно описать при помощи одного предложения, имеющего следующую структуру:

```
<тип элем. массива>[ ] <имя массива> = new <тип элем.> [<кол-во элем.>];
```

Например, рассмотренный выше массив с именем `temp` можно и объявить, и создать при помощи следующего предложения.

```
float[] temp = new float[6];
```

Рис. 8.2 иллюстрирует размещение в памяти компьютера одномерного массива с именем `temp`.

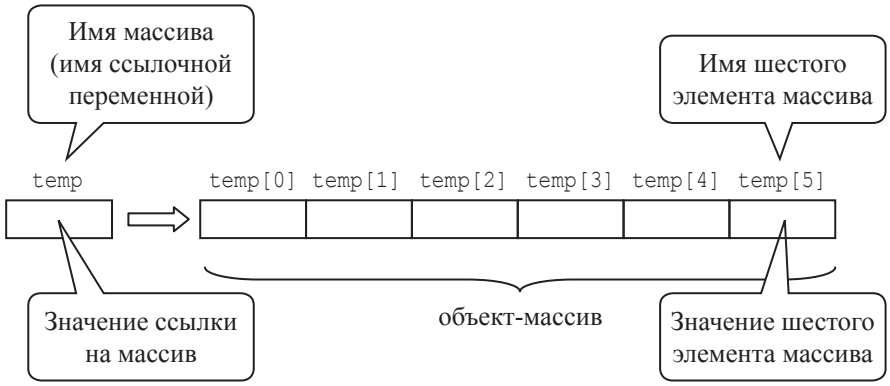



**Рис. 8.2.** Размещение одномерного массива в памяти компьютера

После того, как объявлено имя массива и создан объект-массив, в памяти компьютера размещаются переменная ссылочного типа, хранящая адрес начального участка памяти, в котором располагаются элементы массива, а также все элементы массива. Размер участка памяти, который выделяется для размещения элементов массива, зависит от количества и типа элементов. Адрес конкретного элемента массива можно легко вычислить, если знать адрес начального участка памяти, индекс требуемого элемента и его тип.

После того как массив создан и размещен в памяти компьютера, нельзя изменить количество его элементов ни путем добавления новых элементов, ни путем удаления имеющихся элементов. Можно только присваивать элементам массива новые значения и, таким образом, изменять их значения.

Класс массивов содержит поле с зарезервированным именем `length`, предназначенное только для чтения, которое после создания объекта-массива хранит количество его элементов. Поэтому количество элементов созданного массива можно узнать, обратившись к переменной с именем `length`, при помощи составного имени со следующей структурой.

```
<имя массива> . length
```

На рис. 8.3 приведен пример кода, иллюстрирующий использование поля с именем `length` для получения значения количества элементов массива `temp`.

```
float[] temp = new int[6];  // массив объявлен и создан
int numb = temp.length;     // в переменной numb число 6
```

**Рис. 8.3.** Использование поля `length` для определения количества элементов массива

Элементы массива, созданного при помощи предложения со служебным словом `new`, инициализируются начальными значениями по правилу умолчания в соответствии с их типом. Если элементы массива объявлены как примитивные данные числового типа, то правило умолчания предписывает записать в них нулевые значения. Если элементы массива имеют тип `boolean`, то они инициализируются значениями `false`. Если элементы массива — ссылочные переменные, то они инициализируются значениями `null`.

В ряде случаев можно начинать работу с массивом, элементы которого проинициализированы по правилу умолчания, однако, часто необходимо записать в них определенные начальные значения. Это можно сделать при помощи оператора присваивания.

Если массив состоит из небольшого количества элементов одного из примитивных типов или типа `String`, то можно просто записать несколько предложений, которые присваивают конкретным элементам массива требуемые значения. На рис. 8.4 приведен фрагмент кода, иллюстрирующий этот способ для случая, когда элементами массива являются данные примитивного типа.

```
double[] diameter;          // предложение 1
diameter = new double[4];   // предложение 2
diameter[0] = 19.51;        // предложение 3
diameter[1] = 19.83;        // предложение 4
diameter[2] = 19.05;        // предложение 5
diameter[3] = 20.01;        // предложение 6
```

**Рис. 8.4.** Инициализация массива, состоящего из данных примитивного типа несколькими предложениями с оператором присваивания

Первое предложение на рис. 8.4 объявляет переменную с именем `diameter` ссылочного типа, предназначенную для хранения ссылки на одномерный массив, состоящий из числовых данных типа `double`. Второе предложение создает объект-массив, состоящий из четырех элементов. Сразу же после создания массива все его элементы будут проинициализированы нулевыми значениями в соответствии с правилом умолчания. Предложения 3–6 последовательно заменяют эти нулевые значения требуемыми начальными значениями. Для прямого доступа к элементу массива записывается имя массива, после которого в квадратных скобках указывается значение индекса элемента массива. Индекс первого элемента массива всегда имеет значение 0.

Язык программирования Java позволяет записать код, приведенный на рис. 8.4, значительно более компактно в виде одного предложения.

```
double[] diameter = {19.51, 19.83, 19.05, 20.01};
```

В левой части приведенного предложения объявляется массив с именем `diameter`, состоящий из данных типа `float`. В правой части предложения одновременно детерминируется и количество элементов массива, и их начальные значения. Это делается путем записи списка значений элементов массива в фигурных скобках с разделителем в виде символа «запятая». Первое значение в списке соответствует элементу массива с индексом 0, второй — элементу массива с индексом 1 и т. д.

Аналогичным образом можно инициировать элементы массива, состоящего из данных ссылочного типа. При этом предложения становятся более громоздкими, поскольку необходимо создавать сами объекты, ссылки на которые объединены в массив. На рис. 8.5 приведен фрагмент кода, иллюстрирующий рассматриваемый способ инициализации массива для случая, когда элементами массива являются данные типа `String`.

```
String[] firstName = new String[4];
firstName[0] = new String("Маша");
firstName[1] = new String("Даша");
firstName[2] = new String("Саша");
firstName[3] = new String("Наташа");
```

**Рис. 8.5.** Инициализация массива, состоящего из данных ссылочного типа несколькими предложениями с оператором присваивания

Первое предложение кода, приведенного на рис. 8.5, объявляет и создает массив, состоящий из четырех элементов ссылочного типа `String`. После выполнения этого предложения формируются четыре ссылочные переменные с именами `firstName[0]`, `firstName[1]`, `firstName[2]`, `firstName[3]`, в которые записывается умалчиваемое значение `null`. Последующие предложения создают четыре объекта класса `String` и присваивают в перечисленные ссылочные переменные конкретные значения, сформированные после создания объектов.

Используя нотацию с фигурными скобками, можно вместо кода, приведенного на рис. 8.5, записать одно предложение.

```
String[] firstName = {new String("Маша"),
                      new String("Даша"),
                      new String("Саша"),
                      new String("Наташа")};
```

## 8.2. Начальная инициализация одномерного массива при помощи итерационного процесса

Ясно, что способы инициализации массива, описанные выше, невозможно использовать для инициализации массивов, состоящих из большого количества элементов. В этом случае начальную инициализацию можно выполнить при помощи итерационного процесса, в котором на каждой итерации инициализируется один элемент массива. Поэтому количество итераций равно количеству элементов массива.

Значения, которые присваиваются элементам массива, являются параметрами цикла или их функциями. Ясно, что необходимым условием такого способа начальной инициализации массива является наличие зависимости, позволяющей выразить значение элемента массива через значение параметров цикла.

Для организации цикла, с целью начальной инициализации массива, удобно использовать оператор `for`, поскольку к моменту инициализации известно количество элементов массива и, следовательно, известно количество итераций.

Рассмотрим пример, иллюстрирующий начальную инициализацию массива при помощи итерационного процесса. Задача заключается в инициализации массива целых чисел типа `int`. Массив состоит из 1000 элементов и хранит первые 1000 членов арифметической прогрессии. Первый член прогрессии равен 1, а шаг равен 4. В основе кода, решающего сформулированную задачу, лежит цикл типа «вначале проверка, затем итерация», реализованный на основе оператора `for`. На каждой итерации вычисляется очередной член прогрессии и присваивается в соответствующий элемент массива. Поэтому переменная, хранящая текущее значение членов прогрессии, является параметром цикла. Имеет место простая зависимость между значением параметра цикла и значением соответствующего элемента массива, заключающаяся в том, что значение параметра цикла тождественно значению элемента массива.

На рис. 8.6 приведен вариант кода, решающий описанную задачу.

```
int[] linerSeries;                // предложение 1
linerSeries = new int[1000];      // предложение 2
linerSeries[0] = 1;               // предложение 3
int step = 4;                     // предложение 4
for (int i = 0; i < = linerSeries.length – 1; i++)
   linerSeries[i+1] = linerSeries[i] + step;
```

**Рис. 8.6.** Инициализация массива при помощи итерационного процесса

Первое предложение объявляет массив с именем `linerSeries` (арифметическая прогрессия), состоящий из целых чисел типа `int`. Второе предложение создает массив `linerSeries` с умалчиваемыми значениями элементов. Третье предложение инициализирует первый элемент массива единичным значением, а четвертое — объявляет и

инициализирует целочисленную переменную `step` (шаг). Затем следует заголовок цикла на основе оператора `for`. Условие окончания итераций записано в виде выражения

```
i <= linerSeries.length – 1
```

Для рассматриваемого массива это выражение эквивалентно выражению `i <= 999`. Однако использование в условии окончания итераций поля `length` делает код более универсальным и надежным, поскольку позволяет сколь угодно много раз изменять количество элементов массива во втором предложении, оставляя без изменений условие окончания итераций.

Тело цикла состоит из одного предложения.

```
linerSeries[i+1] = linerSeries[i] + step;
```

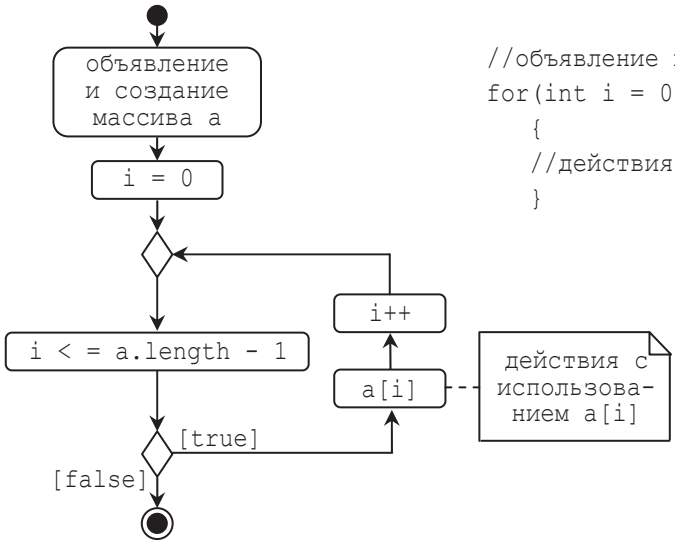
Предложение модифицирует параметр цикла, являющийся одновременно текущим элементом массива. Модификация отражает известную формулу арифметической прогрессии, согласно которой последующий член равен предыдущему члену плюс шаг прогрессии. Спецификой предложения является то, что в качестве переменной-индекса элемента массива используется параметр-счетчик итераций `i`. Это позволяет на каждой итерации получать доступ к очередному элементу массива.

В качестве переменной, хранящей значение индекса элемента массива, может использоваться переменная одного из следующих типов: `byte`, `short`, `int` или `char`. Не допускается использование для этой цели переменной типа `long`. Индексы можно задавать также при помощи выражений, возвращающих значение одного из перечисленных типов, например, `linerSeries[2*i]`, `linerSeries[i%2]`, `linerSeries[++i]`. На этапе выполнения кода может случиться, что индекс примет такое значение, которое превышает максимальное допустимое значение индекса массива. Такая ситуация называется «выход за пределы массива», является исключительной и приводит к прерыванию работы программы.

## 8.3. Оператор цикла `for-each`

В основе кодов, предназначенных для работы с одномерными массивами, лежит цикл типа «вначале проверка условия, затем итерация». Если переменная-индекс элемента массива тождественна параметру-счетчику итераций, то цикл позволяет получить последовательный доступ к каждому элементу массива. Если переменная-индекс элемента массива является функцией от параметра-счетчика, то можно построить цикл, обеспечивающий доступ только к некоторым элементам массива, например, только к элементам с четными значениями индекса. Однако, при кодировании алгоритмов работы с одномерными массивами чаще всего необходимо обеспечить последовательный доступ ко всем элементам массива.

На рис. 8.7 приведена модель цикла, обеспечивающего последовательный доступ ко всем элементам массива и ее отображение в программный код на основе оператора `for`.




```
//объявление и создание массива a
for(int i = 0; i < = a.length – 1; i++)
   {
   //действия с использованием a[i]
   }
```

**Рис. 8.7.** Модель цикла, обеспечивающего последовательный доступ к элементам массива `a` и ее отображение в программный код на основе оператора `for`

На рис. 8.7, переменная `i` является одновременно и переменной-индексом элементов массива, и параметром-счетчиком итераций.

Структура заголовка цикла на основе оператора `for` позволяет программисту «настраивать» итерационный процесс: произвольно инициализировать параметр цикла; произвольно формулировать условие окончания итераций; произвольно модифицировать параметр цикла. Однако, эти возможности являются избыточными в том случае, когда оператор `for` используется для обеспечения последовательного доступа ко всем элементам массива, и могут служить источником логических ошибок. Следствием ошибок при записи заголовка цикла `for` часто является появление исключительной ситуации «выход за пределы массива» на этапе выполнения кода.

В языке программирования Java имеется специальный оператор `for-each` (для каждого), обеспечивающий последовательный доступ ко всем элементам массива. Оператор `for-each` построен таким образом, что исключает логическую ошибку, порождающую исключительную ситуацию «выход за пределы массива». Структура цикла на основе оператора `for-each` приведена на рис. 8.8.

```
for(<тип итерационной пер.> <имя итерационной пер.> : <имя массива>)
   {<тело цикла>}
<последующие предложения>
```

**Рис. 8.8.** Структура цикла на основе оператора `for-each`

В заголовке цикла записывается служебное слово for, поэтому цикл на основе оператора for-each иногда называют усовершенствованным циклом for. После служебного слова for в круглых скобках объявляется итерационная переменная и указывается имя массива, участвующего в итерационном процессе. Разделителем между объявлением итерационной переменной и именем массива является двоеточие. Итерационная переменная объявляется обычным образом, путем указания ее типа и имени. После заголовка цикла размещается тело цикла в виде блока предложений в фигурных скобках. Если тело цикла состоит из одного предложения, то фигурные скобки можно опустить.

Итерационная переменная является буфером, в который в ходе итерационного процесса последовательно помещаются все элементы массива, начиная с первого. Характер итерационной переменной позволяет сделать два важных вывода. Во-первых, тип итерационной переменной должен совпадать или быть совместимым с типом элементов массива. Во-вторых, поскольку в теле цикла действия выполняются не над элементами массива, а только над их копиями, помещенными в итерационную переменную, то цикл на основе оператора for-each не изменяет значения элементов массива. После завершения итерационного процесса значения элементов массива остаются такими же, как и до его начала. Поэтому начальную инициализацию массива нельзя проводить при помощи цикла на основе оператора for-each.

Оператор for-each может использоваться не только для работы с массивами. Его сфера применимости шире и охватывает все типы наборов данных, которые имеются в языке программирования Java.

Рассмотрим несколько примеров, иллюстрирующих работу с одномерными массивами с использованием циклов на основе оператора for-each.

На рис. 8.9 приведен фрагмент кода, иллюстрирующий применение оператора for-each для организации цикла, обеспечивающего суммирование всех элементов массива. Первое предложение кода на рис. 8.9 объявляет, создает и инициализирует массив с именем numbers (числа), состоящий из пяти целых чисел типа int. Второе предложение объявляет и инициализирует целочисленную переменную с именем sum, в которой после выполнения кода должна находиться искомая сумма элементов массива. Далее расположен цикл на основе оператора for-each. В заголовке цикла объявлена итерационная переменная с именем buffer (буфер) и указано имя массива Numbs.

```
int[] numbers = {11, 12, 13, 14, 15};
int sum = 0;
 for(int buffer : numbers){
    System.out.println("Слагаемое: " + buffer);
    sum += buffer;
 }
 System.out.println("Сумма: " + sum);
```

**Рис. 8.9.** Цикл на основе оператора for-each,
в котором обеспечивается доступ ко всем элементам массива

Тело цикла состоит из двух предложений. При помощи предложения

```
System.out.println("Слагаемое: " + buffer);
```

на экран монитора выводится текущее значение переменной `buffer`, которое равно значению текущего элемента массива. При помощи предложения `sum += buffer` предыдущее значение переменной `sum` увеличивается на значение переменной `buffer`. После выхода из цикла выполняется предложение, которое выводит на экран монитора значение переменной `sum`. После завершения выполнения кода на экран выводятся следующие строки:

```
Слагаемое: 11
Слагаемое: 12
Слагаемое: 13
Слагаемое: 14
Слагаемое: 15
Сумма: 65
```

Цикл на основе оператора `for-each` обеспечивает доступ ко всем элементам массива и, следовательно, количество итераций в цикле равно количеству элементов массива. Однако, используя условный оператор `break`, можно прервать итерации в случае выполнения заданного условия. На рис. 8.10 приведен код, полученный из кода на рис. 8.9, и отличающийся от последнего тем, что итерационный процесс прерывается, если сумма, накопленная в переменной `sum`, превышает число 30.

```
int[] numbers = {11, 12, 13, 14, 15};
int sum = 0;
for(int buffer : numbers){
   System.out.println("Слагаемое: " + buffer);
   sum += buffer;
   if(sum > 30) break;          // прерывание итераций, если sum > 30
}
   System.out.println("Сумма: " + sum);
```

**Рис. 8.10.** Цикл на основе оператора `for-each`.
Цикл прерывается, если выполняется заданное условие

После завершения выполнения кода, приведенного на рис. 8.10, на экран монитора выводятся следующие строки.

```
Слагаемое: 11
Слагаемое: 12
Слагаемое: 13
Сумма: 36
```

Оператор for-each следует стараться использовать во всех случаях, когда логика алгоритма требует просмотра всех элементов массива, начиная с первого и заканчивая последним, и сохранения исходного массива в неизменном виде. Рассмотрим еще один пример. Пусть нашей задачей является поиск заданного целого числа в неупорядоченном массиве целых чисел. На рис. 8.11 приведен вариант кода, решающий сформулированную задачу.

```
int[] numbers = {18, 21, 10, 34, 19, 25, 6, 13, 31, 4};
int required;
boolean found = false;
// ввод числа в required
for(int buffer : numbers){
   if(buffer == required){
      found = true;
      break;
   }
}
if(found)
   System.out.println("Найдено");
else
   System.out.println("Не найдено");
```

**Рис. 8.11.** Поиск в неупорядоченном массиве при помощи цикла на основе оператора for-each

Первое предложения кода на рис. 8.11 объявляет, создает и инициализирует одномерный массив, состоящий из десяти чисел типа int. Элементы массива не отсортированы и расположены в произвольном порядке. Второе предложение объявляет переменную с именем required (искомое), которая инициализируется числом, разыскиваемым в массиве. Третье предложение объявляет и инициализирует переменную с именем found (найдено), предназначенную для фиксации факта обнаружения искомого числа. Переменная found инициализируется значением false и принимает значение true, если в массиве обнаруживается искомое число. Затем вводится искомое число. Это место в коде обозначено комментарием.

После объявления и инициализации необходимых данных организуется цикл на основе оператора for-each. В заголовке цикла объявляется переменная с именем buffer, тип которой совпадает с типом элементов массива. Тело цикла состоит из оператора if типа «одно условие — один блок». Условие ветвления оператора if записано в виде булева выражения buffer == required, которое принимает значение true в том случае, когда в массиве обнаруживается искомый элемент. В этом случае выполняется блок оператора if, состоящий из двух предложений. Предложение found = true изменяет значение переменной found, а предложение break прерывает итерационный процесс. Код построен таким образом, что поиск прерывается,

когда при последовательном просмотре элементов массива обнаруживается первый искомый элемент. Если в массиве имеются другие элементы, значения которых равны искомому, то они не будут найдены.

После цикла размещен оператор `if` типа «одно условие — два блока» у которого, в качестве условия ветвления записано имя переменной `found`. Если после завершения итерационного процесса и выхода из цикла переменная `found` принимает значение `true`, то на экран монитора выводится строка со словом «`Найдено`», в противном случае — строка со словами «`Не найдено`».

## 8.4. Базовые алгоритмы работы с одномерными массивами

Одномерные массивы являются простым средством хранения наборов данных, и для решения ряда задач в области императивного программирования удобно использовать одномерные массивы. Алгоритмы решения таких задач, как правило, строятся на основе часто встречающихся алгоритмов, которые мы назовем базовыми алгоритмами работы с одномерными массивами. С некоторыми из базовых алгоритмов мы познакомились, изучая примеры кодов с использованием оператора `for-each`:

- нахождение суммы всех элементов массива, состоящего из числовых данных,
- поиск первого элемента массива с заданным значением.

Рассмотрим еще несколько базовых алгоритмов работы с одномерными массивами и покажем, как они могут использоваться при решении ряда задач императивного программирования. Предметом нашего внимания будут следующие алгоритмы:

- поиск элемента массива с наибольшим/наименьшим значением;
- перестановка элементов массива с заданными индексами;
- циклический сдвиг элементов массива на заданное количество индексов.

### 8.4.1. Поиск элемента массива с наибольшим/наименьшим значением

На рис. 8.12 приведена модель алгоритма, который путем просмотра всех элементов одномерного массива, состоящего из числовых данных, находит и выводит на экран монитора элементы, имеющие наибольшее и наименьшее значения, а также значения индексов этих элементов.

Предполагается, что числовые данные в массиве не дублируют друг друга, не упорядочены и расположены в произвольном порядке.

В начальной части алгоритма располагается несколько структур типа блок. Первая структура объявляет, создает и инициализирует массив с именем

numericalData, состоящий из числовых данных. Последующие две структуры объявляют и инициализируют две переменные с именами max и min. Эти переменные предназначены для хранения значений искомых элементов массива с наибольшим и наименьшим значениями и инициализируются значением первого элемента массива numericalData. Затем расположены еще две структуры, которые объявляют и инициализируют две переменные с именами maxIndex и minIndex. Эти переменные предназначены для хранения индексов элементов массива с наибольшим и наименьшим значениями и инициализируются нулем (индексом первого элемента массива).

Центральная и основная часть алгоритма представляет собой структуру цикл типа «вначале проверка условия, затем итерация» с параметром i. При помощи цикла организуется итерационный процесс, задачей которого является просмотр всех элементов массива. Поэтому переменная-параметр i используется и как переменная-индекс массива.

На каждой итерации значение текущего элемента массива numericalData[i] последовательно сравнивается со значениями переменных max и min. Если текущее значение элемента массива больше, чем значение переменной max, то выполняются два действия. Во-первых, текущее значение элемента массива записывается в переменную max, замещая предыдущее значение этой переменной, а во-вторых, текущий индекс записывается в переменную maxIndex. В противном случае значения max и maxIndex остаются без изменений.

Если текущее значение элемента массива меньше, чем значение переменной min, то выполняются аналогичные два действия. Текущее значение элемента массива записывается в переменную min, а текущий индекс — в переменную minIndex. В противном случае значения переменных min и minIndex остаются без изменений. Действия по сравнению текущего элемента массива с переменными max и min, а также обновление переменных max и min и переменных maxIndex и minIndex моделируются двумя последовательно расположенными структурами развилка типа «одно условие — один блок», из которых формируется итерируемый блок структуры типа цикл.

Ясно, что после завершения всех итераций и выхода из цикла в переменной max находится элемент массива numericalData с наибольшим значением, в переменной min — элемент массива с наименьшим значением, а в переменных maxIndex и minIndex индексы этих элементов. В завершающей части алгоритма расположена структура типа блок, которая выводит на экран монитора значения переменных max, min, maxIndex и minIndex.

Рассмотрим, каким образом модель алгоритма, приведенную на рис. 8.12, можно отобразить в программный код.

Характер алгоритма предполагает просмотр всех элементов массива, с первого — до последнего. Поэтому, как было отмечено выше, цикл, осуществляющий такой просмотр, целесообразно строить на основе оператора for-each. Однако, в заголовке оператора for-each параметр цикла не представлен явно, что исключает возможность оперирования текущими значениями параметра цикла i и запоминать их значения в переменных maxIndex и minIndex. Поэтому при кодировании структуры типа цикл в модели на рис. 8.12 придется использовать оператор for.

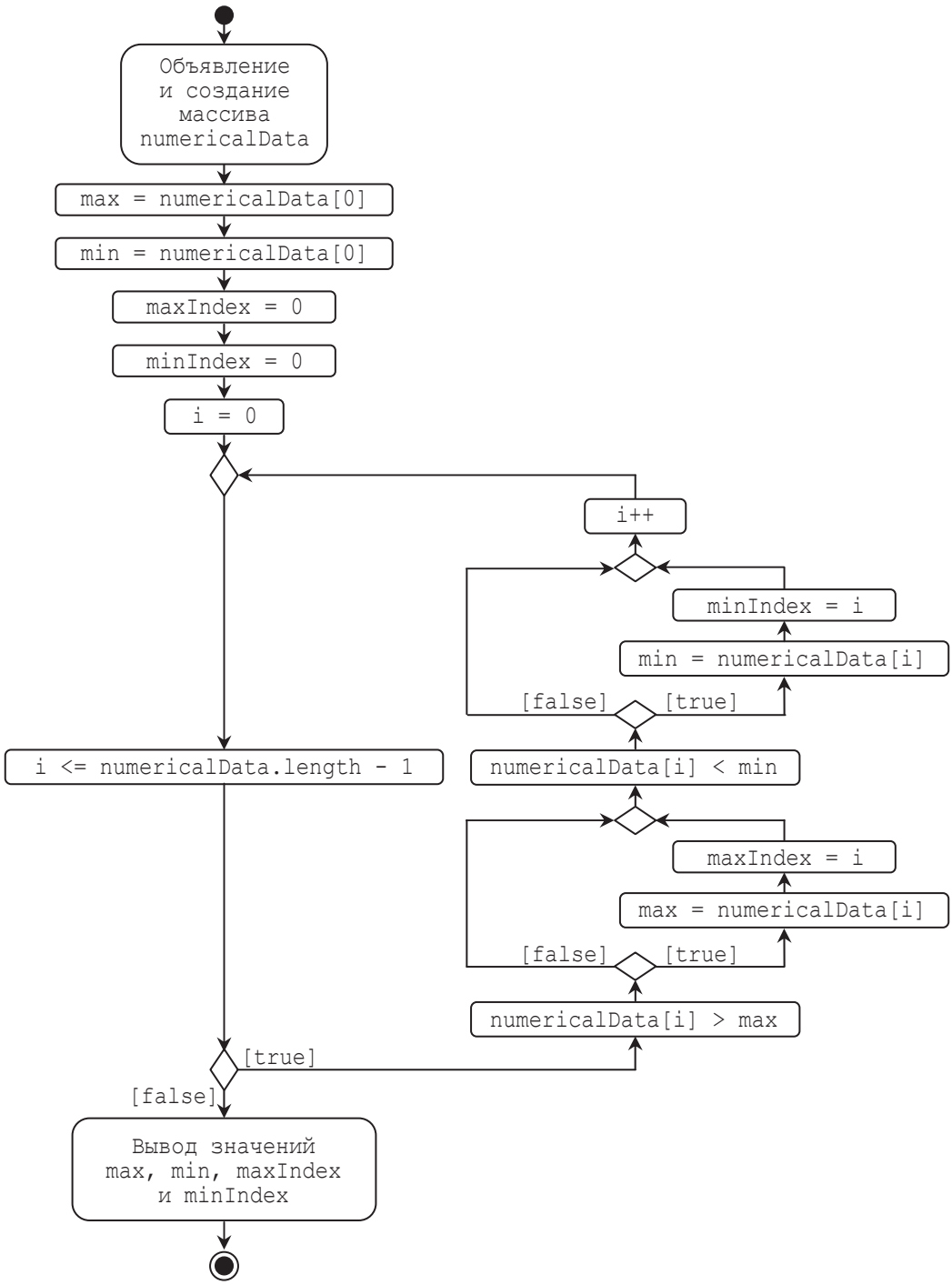



**Рис. 8.12.** Модель алгоритма поиска элементов одномерного массива с наибольшим и наименьшим значениями

На рис. 8.13 приведен вариант кода, соответствующий модели на рис. 8.12.

Первое предложение кода объявляет, создает и инициализирует одномерный массив с именем numericalData, состоящий из десяти целых чисел типа int.

```
int[] numericalData = {18, 21, 10, 34, 19, 25, 6, 13, 31, 4};
int max = numericalData[0],
    min = numericalData[0],
    maxIndex = 0,
    minIndex = 0;
for(int i = 0; i < numericalData.length; i++){
   if(numericalData[i] > max){
      max = numericalData[i];
      maxIndex = i;
   }
   if(numericalData[i] < min){
      min = numericalData[i];
      minIndex = i;
   }
}
System.out.print("Наибольшее значение: " + max);
System.out.println(" имеет элемент с индексом: " + maxIndex);
System.out.print("Наименьшее значение: " + min);
System.out.println(" имеет элемент с индексом: " + minIndex);
```

**Рис. 8.13.** Отображение модели на рис. 8.10 в программный код

Второе предложение, которое занимает пять строк, объявляет и инициализируют переменные max, min, а также переменные maxIndex и minIndex. Переменные max и min инициализируются первым элементом массива numericalData[0], а переменные maxIndex и minIndex — индексом первого элемента массива.

В заголовке цикла на основе оператора for упрощена запись условия окончания итераций, по сравнению с рис. 8.7. Вместо выражения i <= numericalData.length – 1 использовано выражение i < numericalData.length. Легко убедиться, что обе записи формируют одно и то же условие окончания итераций. Тело цикла состоит из двух, последовательно расположенных операторов if типа «одно условие — один блок». Условие ветвления первого оператора if записано в виде булева выражения numericalData[i] > max. Если это выражение принимает значение true, то выполняется блок, состоящий из двух предложений max = numericalData[i] и maxIndex = i. В противном случае проверяется условие ветвления второго оператора if, которое записано в виде булева выражения numericalData[i] < min. Если это выражение принимает значение true, то выполняется блок, состоящий из предложений min = numericalData[i] и minIndex = i.

Последние четыре предложения выводят на экран монитора найденные значения элементов массива и их индексы в виде следующих двух строк:

```
Наибольшее значение: 34 имеет элемент с индексом: 3
Наименьшее значение: 4 имеет элемент с индексом: 9
```

Несмотря на то, что характер рассмотренного алгоритма поиска предполагает просмотр всех элементов массива, мы не смогли использовать для организации этого просмотра оператор `for-each`, поскольку характер задачи требовал запоминать значения индексов элементов массива. Если снять это требование и ограничиться поиском только элементов массива с наибольшим и наименьшим значениями, то можно организовать просмотр элементов массива при помощи оператора `for-each`.

На рис. 8.14 приведен фрагмент кода, решающий задачу поиска элементов массива с наибольшим и наименьшим значениями, в котором просмотр элементов массива осуществляется циклом на основе оператора `for-each`. Цикл на основе оператора `for-each` не соответствует в точности модели, приведенной на рис. 8.7, поскольку в заголовке оператора `for-each` параметр цикла не представлен явно. Поэтому при отображении моделей в программный код с оператором `for-each` следует опускать действия по инициализации и модификации параметра цикла `i`, помня, что оператор `for-each` делает это автоматически.

```
int[] numericalData = {18, 21, 10, 34, 19, 25, 6, 13, 31, 4};
int max = numericalData[0],
    min = numericalData[0];
for(int current : numericalData){
   if(current > max)
      max = current;
   if(current < min)
      min = current;
}
System.out.println("Наибольшее значение: " + max);
System.out.println("Наименьшее значение: " + min);
```

**Рис. 8.14.** Поиск элементов массива с наибольшим и наименьшим значениями при помощи цикла на основе оператора `for-each`

Первое предложение кода объявляет, создает и инициализирует одномерный массив с именем `numericalData`, состоящий из десяти целых чисел типа `int`. Второе — объявляет две переменные `max` и `min`, которые инициализируются значением первого элемента массива `numericalData[0]`.

В заголовке цикла на основе оператора `for-each` объявляется итерационная переменная с именем `current`, тип которой совпадает с типом элементов массива.

Тело цикла состоит из двух, последовательно расположенных операторов `if` типа «одно условие — один блок». Условие ветвления первого оператора `if` записано в виде булева выражения `current > max`. Если оно принимает значение `true`, то выполняется предложение `max = current`. В противном случае проверяется условие ветвления второго оператора `if`, которое записано в виде булева выражения `current < min`. Если это выражение принимает значение `true`, то выполняется предложение `min = current`.

Последние два предложения выводят на экран монитора значения переменных `max` и `min`.

При решении задачи поиска элементов массива с наибольшим и наименьшим значением, а также значений их индексов может случиться так, что в массиве имеется несколько элементов с наибольшим либо наименьшим значением. Ясно, что в этом случае значения самих элементов массива будут одинаковы, однако значения их индексов будут разными. Если массив может содержать *несколько элементов с наибольшим либо наименьшим значениями*, то задача поиска значений индексов этих элементов должна быть уточнена одним из следующих способов: (1) найти индекс первого элемента с наибольшим/наименьшим значением; (2) найти индекс последнего элемента с наибольшим/наименьшим значением; (3) найти индексы всех элементов с наибольшим/наименьшим значением.

Модель алгоритма поиска, приведенная на рис. 8.12, осуществляет *поиск индекса первого элемента* массива с наибольшим/наименьшим значением. Это обусловлено тем, что в обеих структурах типа развилка (см. рис. 8.10) условия ветвления записаны в виде строгих неравенств: `numericalData[i] > max` и `numericalData[i] < min`. Следовательно, значения переменных `maxIndex` или `minIndex` обновляются только в том случае, когда текущий элемент массива строго больше/меньше того, который уже хранится в переменной `max` или `min`. Поэтому, если встречается текущий элемент массива, равный тому, который уже хранится в `max` или в `min`, то значения переменных `maxIndex` или `minIndex` останутся неизменными.

Легко модифицировать эту модель для *поиска индекса последнего элемента* массива с наибольшим/наименьшим значением. Для этого необходимо только записать условия ветвления в виде `numericalData[i] >= max` и `numericalData[i] <= min`. Теперь значения переменных `maxIndex` или `minIndex` будут обновляться и в том случае, когда текущий элемент массива равен тому, который хранится в переменной `max` или `min`. Поэтому, если при «просмотре» элементов массива встретится еще один, имеющий наибольшее/наименьшее значение, то это вызовет обновление переменных `maxIndex` или `minIndex`.

Решение задачи *поиска всех индексов элементов* массива с наибольшим/наименьшим значением сложнее, поскольку для запоминания множества индексов необходим еще один массив.

Рассмотрим пример решения задачи *поиска индекса первого элемента массива с наибольшим значением и индекса последнего элемента с наименьшим значением*. Задача заключается в следующем. Пусть имеется массив значений средних дневных температур воздуха в декабре месяце: $t_1$, $t_2$, ..., $t_{31}$. Необходимо определить день,

когда первый раз была зарегистрирована наименьшая температура, и день, когда последний раз была зарегистрирована наибольшая температура.

Ясно, что массив температур имеет 31 элемент, а номера дней соответствуют индексам элементов массива температур. Например, «1 декабря» соответствует элементу массива температур с индексом 0, «2 декабря» — элементу массива с индексом 1 и т. д.

На рис. 8.15 приведена модель алгоритма решения сформулированной задачи. Структура алгоритма, приведенного на рис. 8.15, аналогична структуре алгоритма на рис. 8.12.

В начальной части алгоритма располагается несколько структур типа блок, при помощи которых выполняются следующие действия. Объявляется, создается и инициализирует массив с именем `temperature`, состоящий из 31 элемента и хранящий значения средних дневных температур в декабре. Объявляются и инициализируют две переменные с именами `maxT` и `minT`, предназначенные для хранения значений наибольшей и наименьшей из зарегистрированных температур. Переменные инициализируются значением первого элемента массива `temperature`. Объявляются и инициализируют две переменные с именами `maxTIndex` и `minTIndex`, предназначенные для хранения первого индекса элементов массива с наибольшим значением и последнего индекса элемента с наименьшим значением. Переменные инициализируются индексом первого элемента массива температур.

Центральная часть алгоритма представляет собой структуру цикл типа «вначале проверка условия, затем итерация» с параметром `i`, который одновременно является переменной-индексом массива. Итерируемый блок состоит из двух, последовательно расположенных структур развилка типа «одно условие — одно действие», при помощи которых на каждой итерации значение текущего элемента массива `temperature` последовательно сравнивается со значениями переменных `maxT` и `minT`.

Если текущее значение элемента массива строго больше, чем значение переменной `maxT`, то выполняются два действия. Во-первых, текущий элемент массива записывается в переменную `maxT`, замещая предыдущее значение этой переменной, а во-вторых, текущий индекс записывается в переменную `maxTIndex`. Поскольку отмеченные действия выполняются в том случае, когда значение текущего элемента массива температур строго больше температуры, хранящейся в `maxT`, то переменная `maxTIndex` хранит значение *индекса первого элемента массива температур с наибольшим значением*.

Если текущее значение элемента массива меньше или равно значению переменной `minT`, то выполняются аналогичные действия. Текущее значение элемента массива записывается в переменную `minT`, а текущий индекс — в переменную `minTIndex`. Поскольку переменная `minTIndex` обновляется при каждом новом наименьшем значении элемента массива температур, то после завершения всех итераций она будет хранить *индекс последнего элемента массива температур с наименьшим значением*.

В завершающей части алгоритма расположена структура типа блок, которая выводит на экран монитора номер первого дня, когда зарегистрирована наибольшая

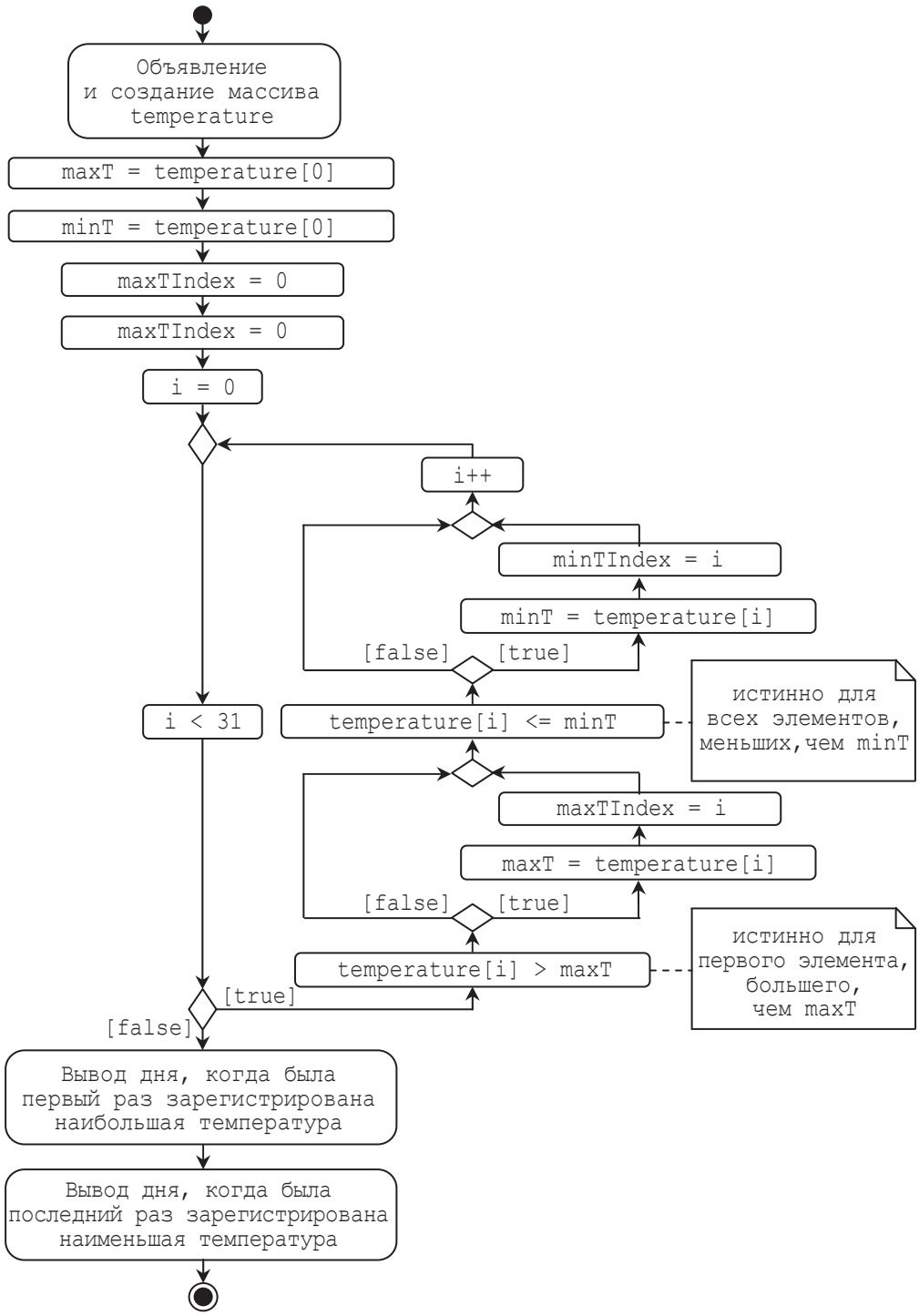



**Рис. 8.15.** Модель алгоритма поиска первого наибольшего и последнего наименьшего элементов одномерного массива

температура, и номер последнего дня декабря, когда зарегистрирована наименьшая температура.

На рис. 8.16 приведено отображение модели на рис. 8.15 в программный код. Первое предложение кода объявляет и создает одномерный массив с именем `temperature`, состоящий из 31 числа типа `float`.

Второе предложение объявляет и инициализирует переменные `maxT`, `minT`, которые инициализируются первым элементом массива `temperature`. Третье предложение объявляет и инициализирует нулевыми значениями две целочисленные переменные `maxTIndex` и `minTIndex`.

Затем комментарием отмечено то место кода, где осуществляется инициализация массива `temperature`.

В заголовке цикла на основе оператора `for` объявляется и инициализируется нулем переменная-параметр цикла `i` специфицируется условие окончания итераций выражением `i < 31`, а также выражение, модифицирующее параметр цикла `i++`.

```
float[] temperature = new float[31];
float maxT = temperature[0],
      minT = temperature[0];
int maxTIndex = 0,
    minTIndex = 0;
// инициализация массива temperature
for(int i = 0; i < 31; i++){
   if(temperature[i] > maxT){
      maxT = temperature[i];
      maxTIndex = i;
   }
   if(temperature[i] <= minT){
      minT = temperature[i];
      minTIndex = i;
   }
}
int firstDay = maxTIndex +1,
    lastDay = minTIndex + 1;
System.out.print("Наибольшая температура: " + maxT);
System.out.println(" наблюдалась первый раз: " + firstDay + " декабря");
System.out.print("Наименьшая температура: " + minT);
System.out.println(" наблюдалась последний раз: " + lastDay + " декабря");
```

**Рис. 8.16.** Отображение модели на рис. 8.15 в программный код

Тело цикла состоит из двух, последовательно расположенных операторов `if` типа «одно условие — один блок». Условие ветвления первого оператора `if` записано в виде булева выражения `temperature[i] > maxT`. Если это

выражение принимает значение `true`, то выполняется блок, состоящий из двух предложений `maxT = temperature[i]` и `maxTIndex = i`. В противном случае проверяется условие ветвления второго оператора `if`, которое записано в виде выражения `temperature[i] <= minT`. Если это выражение принимает значение `true`, то выполняется блок, состоящий из предложений `minT = temperature[i]` и `minTIndex = i`.

После выхода из цикла объявляются и инициализируются две целочисленные переменные с именами `firstDay` и `lastDay`. Эти переменные упрощают запись предложений, осуществляющих вывод на экран монитора результатов работы кода.

Последние четыре предложения выводят на экран монитора найденные наибольшие и наименьшие значения температур и соответствующие им дни месяца.

### 8.4.2. Перестановка элементов массива

Элементы массива размещены в смежных участках памяти. К участку памяти, в котором размещен элемент массива, возможен независимый доступ, а индекс массива можно рассматривать как символический адрес этого участка памяти. Для упрощения дальнейшего изложения назовем адресуемые участки памяти, предназначенные для размещения элементов массива, ячейками.

Простейшая задача перестановки элементов одномерного массива заключается в обмене содержимым двух ячеек, имеющих различные индексы `i` и `j`. Очевидно, что для выполнения такого обмена без потери информации необходима буферная ячейка, в которую временно перемещается содержимое одной из ячеек, участвующих в обмене. Рис. 8.17 иллюстрирует последовательность действий при решении простейшей задачи перестановки элементов одномерного массива с использованием буферной ячейки.

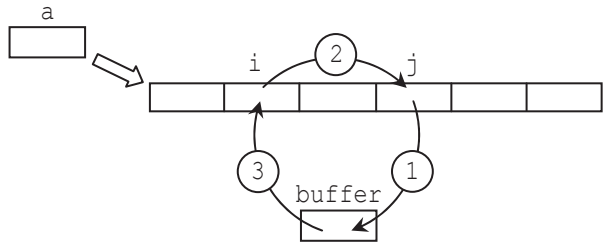



**Рис. 8.17.** Последовательность действий при перестановке двух элементов одномерного массива с использованием буферной ячейки

На рис 8.17 изображен массив с именем `a`, выделены два его элемента `a[i]` и `a[j]`, а также буферная ячейка с именем `buffer`. Обмен значениями элементов массива `a[i]` и `a[j]` осуществляется в три этапа.

На первом этапе один из элементов массива (например, `a[j]`) перемещается в буферную ячейку. В программном коде это действие кодируется при помощи предложения с оператором присваивания.

```
buffer = a[j];
```

После выполнения первого этапа элемент массива `a[j]` хранится в двух местах, в ячейке массива с адресом `j` и в буферной ячейке.

На втором этапе элемент массива `a[i]` перемещается в ячейку массива с адресом `j`, замещая в ней предыдущее содержимое — элемент массива `a[j]`. В программном коде это действие кодируется при помощи следующего предложения:

```
a[j] = a[i];
```

После выполнения второго этапа элемент массива `a[i]` находится в двух ячейках с адресами `i` и `j`.

На третьем этапе содержимое буферной ячейки перемещается в ячейку массива с адресом `i`. В программном коде это действие кодируется при помощи предложения

```
a[i] = buffer;
```

После выполнения третьего этапа завершается перестановка элементов массива `a[i]` и `a[j]`, а в буферной ячейке остается элемент массива `a[i]`.

Рассмотренный способ решения простейшей задачи перестановки элементов одномерного массива лежит в основе алгоритмов решения более сложных задач перестановки. Например, поменять местами элементы массива с четными и нечетными значениями индекса.

Рассмотрим *решение задачи инвертирования массива*, или перестановки элементов массива в обратном порядке. Имеется в виду такая перестановка, когда первый элемент массива становится последним, второй — предпоследним и т. д.

На рис. 8.18 приведена модель алгоритма инвертирования одномерного массива.

Для инвертирования одномерного массива `a` на основе рассмотренного способа перестановки его двух элементов (см. рис. 8.17) необходима организация итерационного процесса на основе цикла типа «вначале проверка условия, затем итерация». Первая итерация выполнит перестановку первого и последнего элементов массива, вторая — перестановку второго и предпоследнего элементов массива и т. д. Параметрами цикла будут переменные-индексы `i` и `j`, которые инициализируются значениями 0 и `a.length - 1` соответственно. Модификация параметров заключается в увеличении значения `i` на 1 и в уменьшении значения `j` на 1. Если массив содержит нечетное количество элементов, то условием окончания итерационного процесса будет совпадение значений индексов `i` и `j`. Если массив содержит четное количество элементов, то условием окончания итерационного процесса будет истинность неравенства `i > j`.

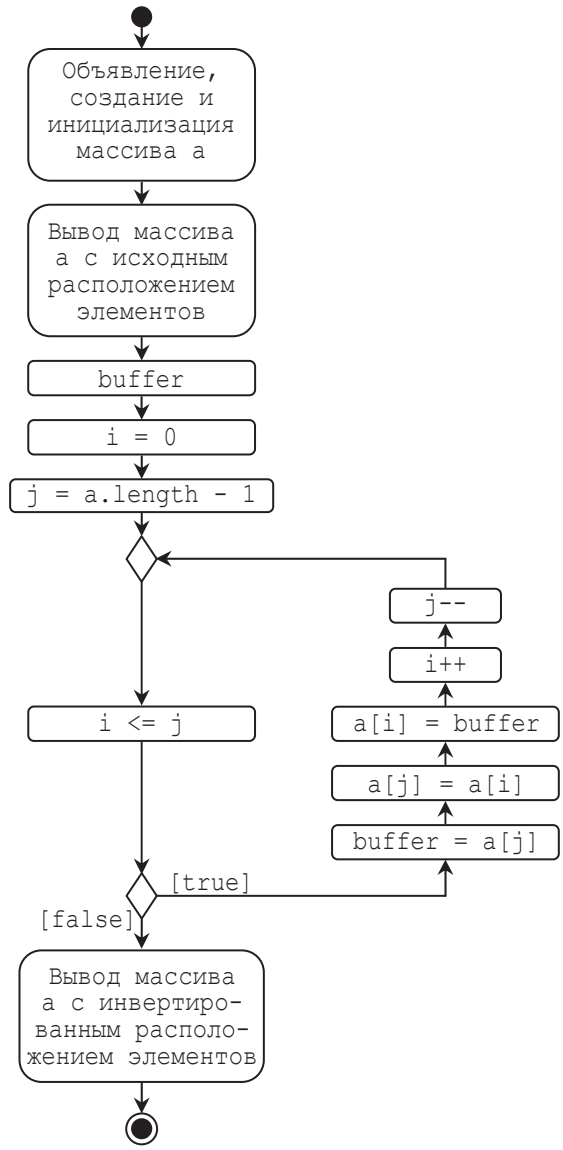



**Рис. 8.18.** Модель алгоритма инвертирования одномерного массива

В начальной части алгоритма расположены несколько структур типа блок. Алгоритм начинается с объявления, создания и инициализации массива с именем `a`. Затем исходный массив выводится на экран монитора и объявляется переменная с именем `buffer`, предназначенная для временного хранения одного из элементов массива во время перестановки. После этого объявляются две переменные `i` и `j`, которые «работают» и как переменные-индексы массива, и как переменные-параметры цикла. Переменная `i`, рассматриваемая как переменная-индекс массива, «движется» от первого элемента массива к последнему и поэтому инициализируется значением

индекса первого элемента массива. Переменная `j`, как переменная-индекс массива, «движется» от последнего элемента массива к первому и инициализируется значением индекса последнего элемента.

Центральной частью алгоритма является структура цикл типа «вначале проверка, затем итерация». Условие окончания итераций записано в виде булева выражения

```
i <= j,
```

которое учитывает условия окончания итераций для массива, состоящего как из нечетного, так и из четного количества элементов.

Тело цикла состоит из пяти последовательно расположенных структур типа блок. Первые три выполняют действия, необходимые для перестановки элементов массива `a[i]` и `a[j]` с использованием буферной переменной `buffer`. Четвертая и пятая осуществляют модификацию переменных-параметров цикла `i` и `j`.

После завершения всех итераций и выхода из цикла целесообразно вывести на экран монитора массив в инвертированном виде.

На рис. 8.19 приведен вариант отображения модели на рис. 8.18 в программный код. Детали начальной инициализации массива, а также вывода его значений на экран монитора опущены.

```
int[] a[] = new a[10];
int buffer;
// инициализация массива a
// вывод исходного массива a
   for(int i = 0, j = a.length -1; i <= j; i++, j--){
      buffer = a[j];
      a[j] = a[i];
     a[i] = buffer;
   }
// вывод инвертированного массива
```

**Рис. 8.19.** Отображение модели на рис. 8.18 в программный код

Особенностью кода, приведенного на рис. 8.19, является использование нотации оператора `for`, которая позволяет в заголовке цикла объявлять, инициализировать и модифицировать несколько параметров цикла (см. 7.9). Это позволяет записать код в понятном и компактном виде. Тело цикла содержит только три предложения с операторами присваивания, которые кодируют действия по перестановке элементов массива `a[i]` и `a[j]` с использованием буферной переменной `buffer`.

### 8.4.3. Циклический сдвиг элементов массива

В ряде случаев приходится циклически сдвигать элементы одномерного массива влево или вправо. Циклический сдвиг предполагает, что все *ячейки массива соединены в кольцо* и после ячейки с последним элементом следует ячейка с первым элементом. При циклическом сдвиге одномерного массива на одну ячейку вправо каждый элемент, кроме последнего, перемещается в ближайшую правую ячейку, а последний элемент перемещается на место первого. При циклическом сдвиге одномерного массива на одну ячейку влево каждый элемент, кроме первого, перемещается в ближайшую левую ячейку, а первый элемент перемещается на место последнего. Алгоритм циклического сдвига массива может строиться на использовании буферной ячейки, в которую на первом этапе временно записывается последний (при сдвиге вправо) либо первый (при сдвиге влево) элемент массива. Затем все элементы последовательно перезаписываются в смежные ячейки, расположенные справа или слева, в зависимости от направления сдвига. Завершается алгоритм перенесением содержимого буфера в первую (при сдвиге вправо) или последнюю (при сдвиге влево) ячейку массива.

Рис. 8.20 иллюстрирует идею алгоритма циклического сдвига элементов одномерного массива. На рис. 8.20 изображен одномерный массив с именем `a`, состоящий из шести элементов, и буферная ячейка с именем `buffer` для временного хранения последнего либо первого элементов массива. Номерами отмечена последовательность действий, которая должна выполняться алгоритмом циклического сдвига. В левой части рисунка изображена последовательность действий при циклическом сдвиге вправо, а в правой части — при циклическом сдвиге влево.

Действие 1 сохраняет последний либо первый элемент массива в буфере на время перезаписи элементов массива в смежные ячейки.

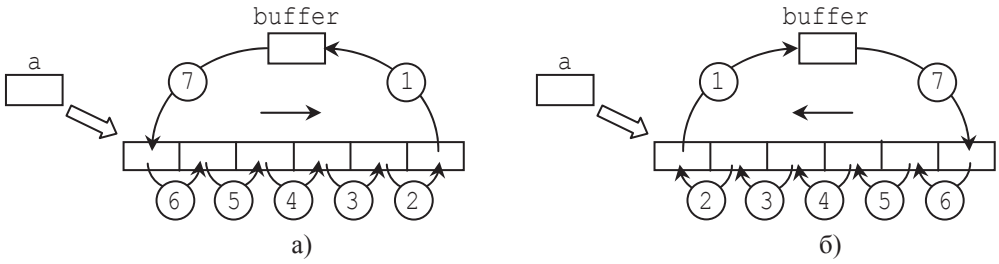



**Рис. 8.20.** Последовательность действий при циклическом сдвиге элементов массива на одну ячейку. а) сдвиг вправо; б) Сдвиг влево

Действия 2–6 обеспечивают последовательную перезапись элементов массива в смежные ячейки без потери информации. Действие 7 возвращает элемент массива, хранящийся в буфере в первую либо последнюю ячейку.

Модели алгоритмов циклического сдвига элементов одномерного массива на одну ячейку вправо и влево приведены на рис. 8.21.

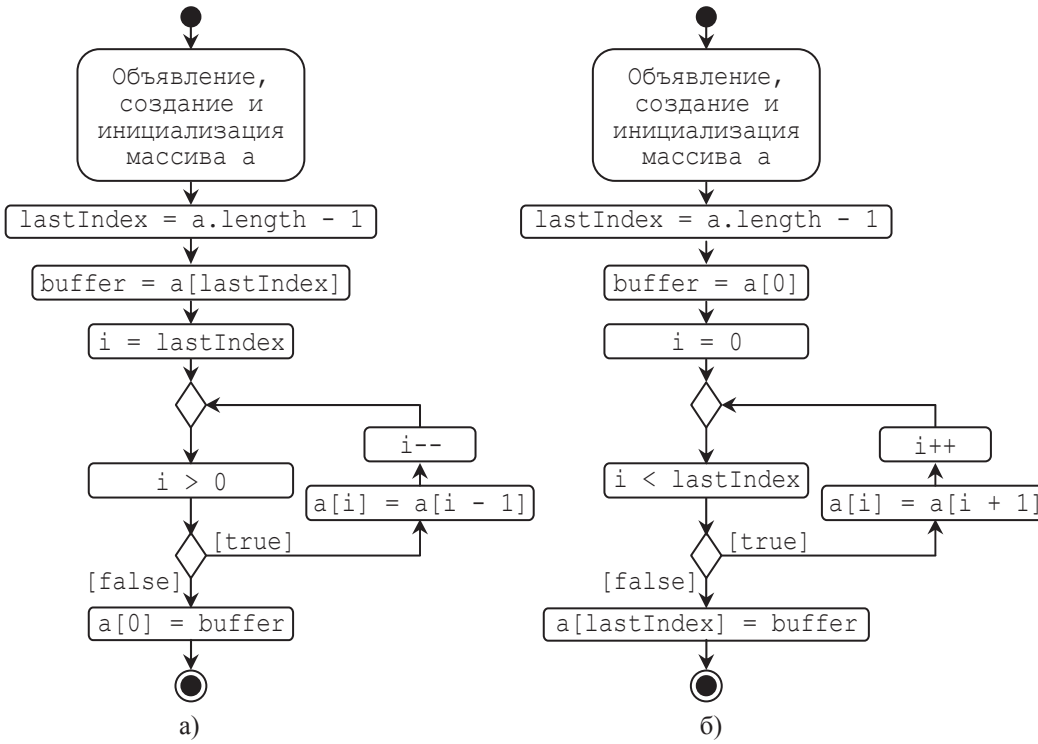



**Рис. 8.21.** Модели алгоритмов циклического сдвига одномерного массива на одну ячейку. а) сдвиг вправо; б) сдвиг влево

Модели, изображенные на рис. 8.21, структурно идентичны и отличаются действиями, выполняемыми некоторыми структурами типа блок, входящими в их состав.

Опишем модель алгоритма циклического сдвига одномерного массива на одну ячейку вправо, изображенную на рис. 8.21 а). В начальной части модели последовательно расположены четыре структуры типа блок, выполняющие следующие действия: (1) объявление, создание и начальная инициализация одномерного массива с именем `a`; (2) объявление переменной с именем `lastIndex` (последний индекс) и ее инициализация индексом последнего элемента массива; (3) объявление переменной с именем `buffer` и ее инициализация последним элементом массива; (4) объявление переменной с именем `i` (являющейся одновременно и переменной-параметром цикла, и переменной-индексом массива) и инициализация этой переменной значением индекса последнего элемента массива.

В центральной части алгоритма расположена структура цикл типа «вначале проверка, затем итерация». На каждой итерации последовательно выполняются две структуры типа блок. Первый блок перезаписывает текущий элемент массива в ближайшую смежную ячейку, расположенную справа, а второй — модифицирует параметр цикла (индекс массива) путем уменьшения его на единицу. Итерации продолжаются до тех пор, пока параметр цикла (индекс массива) строго больше нуля.

После структуры типа цикл в завершающей части алгоритма расположена структура типа блок, которая переносит содержимое переменной `buffer` на место первого элемента массива.

На рис. 8.22 приведен фрагмент кода, в который может быть отображена модель, приведенная на рис. 8.21.

```
int[] a = {2, 4, 6, 8, 10, 12};
int lastIndex = a.length – 1;
int buffer = a[lastIndex];
for(int i = lastIndex; i > 0; i--)
   a[i] = a[i - 1];
a[0] = buffer;
```

**Рис. 8.22.** Отображение модели на рис. 8.21 а) в программный код

Первое предложение кода на рис. 8.22 объявляет, создает и инициализирует одномерный массив с именем `a`, состоящий из шести целых чисел.

Второе предложение объявляет переменную с именем `lastIndex` и присваивает ей индекс последнего элемента массива, который определяется путем вычитания единицы из количества элементов в массиве при помощи выражения.

```
a.length – 1
```

Третье предложение объявляет переменную с именем `buffer` и присваивает ей значение последнего элемента массива.

Затем следует заголовок цикла, организованного на основе оператора `for`. В заголовке цикла объявляется переменная-параметр цикла с именем `i`, которая инициализируется индексом последнего элемента массива, специфицируется условие окончания итераций в виде булева выражения `i > 0` и выражение `i--`, при помощи которого модифицируется параметр цикла.

Тело цикла состоит из одного предложения.

```
a[i] = a[i - 1];
```

При помощи этого предложения на каждой итерации значение элемента массива с индексом `i - 1` присваивается элементу массива, имеющему индекс `i`, (ближайшему элементу, расположенному справа). Переменная-индекс массива изменяет свое значение синхронно с переменной-параметром цикла.

Последнее предложение выполняется после выхода из цикла и присваивает первому элементу массива значение переменной `buffer`.

Ясно, что на основе рассмотренных алгоритмов циклического сдвига элементов одномерного массива на одну ячейку можно строить алгоритмы сдвига элементов массива на произвольное, наперед заданное, количество ячеек.

## 8.5. Сортировка одномерных массивов

Сортировкой массива называется расположение элементов массива в порядке возрастания или убывания их значений. Сортировке могут подвергаться массивы с элементами такого типа, который позволяет сравнивать значения элементов и в результате сравнения устанавливать между любыми двумя элементами массива отношение «больше» или «меньше». К таким типам относятся:

1. все числовые типы (`byte`, `short`, `int`, `long`, `float`, `double`), поскольку между любыми двумя числами можно установить отношение «больше» или «меньше»;
2. символьный тип `char`, поскольку любой литерал типа `char` представим в виде Unicode-номера;
3. строковый тип `String`, поскольку строки состоят из символов, а символы представимы Unicode-номерами;
4. ссылочные типы, определяемые классом, если в классе реализован предопределенный интерфейс `Comparable` (сопоставимый).

Предложено и описано большое количество методов и алгоритмов сортировки одномерных массивов, и, наверное, существует примерно столько же методов и алгоритмов, которые не нашли своего описания. Полезно, прежде чем изучать приведенные ниже алгоритмы, *самостоятельно изобрести* несколько методов сортировки одномерных массивов. Изобразите на листе бумаги одномерный массив, состоящий из небольшого количества целых положительных чисел, расположенных в произвольном порядке. Затем, комбинируя базовые алгоритмы работы с одномерным массивом, попробуйте изобрести способ размещения элементов этого массива в порядке возрастания их значений. Напомним, что к базовым алгоритмам работы с одномерным массивом мы отнесли: (1) поиск элемента массива с наибольшим/наименьшим значением и его индекса; (2) перестановка элементов массива с заданными индексами; (3) сдвиг элементов массива на заданное количество индексов. Если такая работа будет проделана, то среди изобретенных методов будут либо известные классические методы сортировки, либо их модификация, либо новые и не описанные в литературе методы сортировки.

В настоящем подразделе рассмотрены следующие, ставшие классическими, методы сортировки одномерных массивов:

- сортировка методом «пузырька»;
- сортировка методом прямого выбора;
- сортировка методом вставки.

Перечисленные методы сортировки выбраны, поскольку они отображаются в простые и компактные коды, однако обладают тем недостатком, что порождают большое количество операций процессора при их выполнении. Поэтому перечисленные методы эффективны для сортировки массивов, состоящих из небольшого количества элементов.

### 8.5.1. Сортировка методом «пузырька»

Рассмотрим идею сортировки массива методом пузырька на примере сортировки одномерного массива, состоящего из пяти целых чисел. Рис. 8.23 иллюстрирует последовательность действий над элементами массива при использовании сортировки методом «пузырька». Числа в массиве расположены в произвольном порядке, и задача заключается в таком упорядочивании чисел, чтобы в итоге они расположились в возрастающем порядке.

На рис. 8.23 графический символ массива изображен вертикально. Вверху располагается ячейка первого элемента массива, а внизу — последнего. Это сделано для того, чтобы пояснить наименование метода. По мере сортировки (при просмотре рисунка слева направо) видно, что текущие элементы массива с б*о*льшими значениями «опускаются» вниз, а текущие элементы массива с меньшими значениями «поднимаются вверх», как это делает пузырек воздуха в вертикальном цилиндре с водой.

Для того, чтобы отсортировать массив методом «пузырька», он должен быть просмотрен несколько раз. При первом просмотре обнаруживается элемент массива, имеющий наибольшее значение, и «опускается» в последнюю ячейку массива. При втором просмотре из не отсортированных элементов массива опять выбирается элемент с наибольшим значением и «опускается» в предпоследнюю ячейку массива. На рис. 8.23 элементы массива, которые заняли свое место в отсортированном массиве, изображены жирным шрифтом. Легко увидеть, что для того, чтобы отсортировать массив, состоящий из `n` элементов, методом «пузырька» необходимо просмотреть массив `n-1` раз.

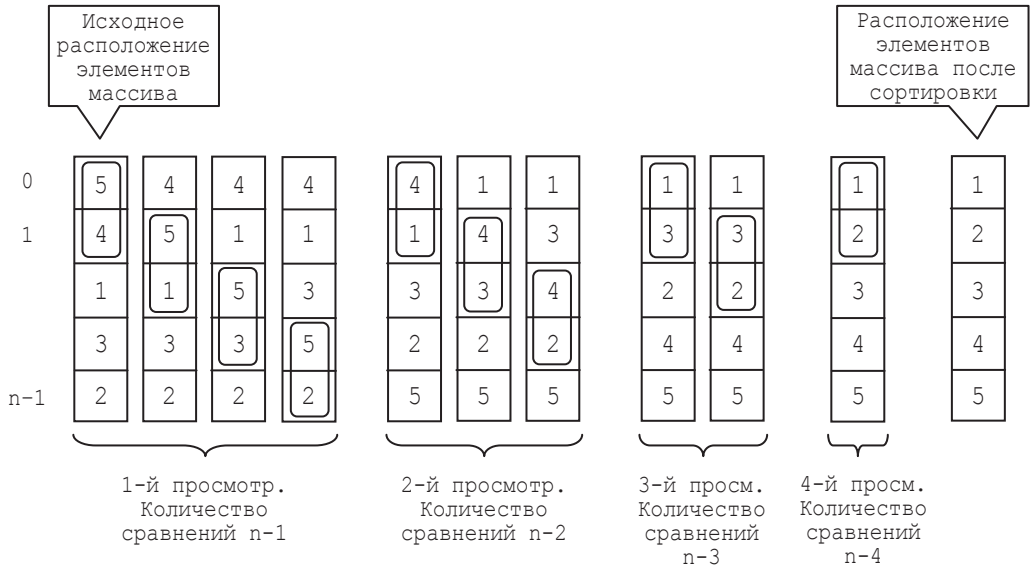



**Рис. 8.23.** Последовательность действий при сортировке одномерного массива методом «пузырька»

При каждом просмотре последовательно сравниваются значения пары элементов массива, *находящихся в смежных ячейках*. Сравнение начинается с пары элементов, находящихся в ячейках с индексами 0 и 1. Если обнаруживается, что элемент, имеющий большее значение, находится в ячейке с меньшим индексом, то выполняется перестановка этих двух элементов. На рис. 8.23 при первом просмотре в ячейке с индексом 0 находится число 5, а в ячейке с индексом 1 — число 4 (крайний левый графический символ массива). Следовательно, после сравнения этих чисел должна быть выполнена их перестановка. Затем сравнению подвергаются числа, находящиеся в ячейках с индексами 1 и 2, и при необходимости выполняется перестановка элементов массива, находящихся в этих ячейках. Общее количество сравнений при первом просмотре массива равно `n-1`. При последующем просмотре сравнение опять начинается с ячеек, имеющих индексы 0 и 1. При втором «просмотре» количество сравнений равно `n-2`, при третьем «просмотре» — `n-3` и т. д. При каждом последующем просмотре количество сравнений уменьшается на единицу.

Представим теперь идею сортировки массива методом пузырька в виде модели алгоритма, составленной из типовых поведенческих структур (см. рис. 7.26). Центральной частью этого алгоритма будут вложенные циклы, в которых итерации внутреннего цикла обеспечивают сравнение текущей пары смежных элементов массива и их перестановку, а итерации внешнего цикла обеспечивают текущие просмотры массива.

Модель алгоритма сортировки одномерного массива в порядке возрастания значений элементов массива методом «пузырька» приведена на рис. 8.24.

В начальной части алгоритма расположены структуры типа блок, при помощи которых объявляется, создается, инициализируется и выводится на экран монитора одномерный массив. Затем располагаются еще несколько блоков. Первый объявляет и инициализирует целочисленную переменную с именем `numbOfCompar` (количество сравнений). Переменная `numbOfCompar` предназначена для хранения количества сравнений смежных элементов массива при каждом его просмотре. При первом просмотре массива количество сравнений наибольшее и на единицу меньше количества элементов массива. Поэтому переменная `numbOfCompar` инициализируется выражением `a.length - 1`. Перед каждым последующим просмотром значение переменной `numbOfCompar` уменьшается на единицу.

Последовательность блоков завершается блоком, объявляющим буферную ячейку с именем `buffer`, необходимую для перестановки элементов массива.

Центральная часть алгоритма представляет собой вложенные циклы, в которых и внешний, и внутренний циклы относятся к типу «вначале проверка, затем итерация». Параметром внешнего цикла является переменная `i`, которая является, по сути, счетчиком просмотров массива. Поэтому параметр `i` инициализируется единицей и перед каждой последующей итерацией увеличивается на единицу. Условием окончания количества итераций внешнего цикла (количества просмотров) является истинность булева выражения `i <= a.length - 1`. Итерируемый блок внешнего цикла составлен из внутреннего блока типа цикл, после которого следует блок, уменьшающий на единицу значение переменной `numbOfCompar`. Задачей

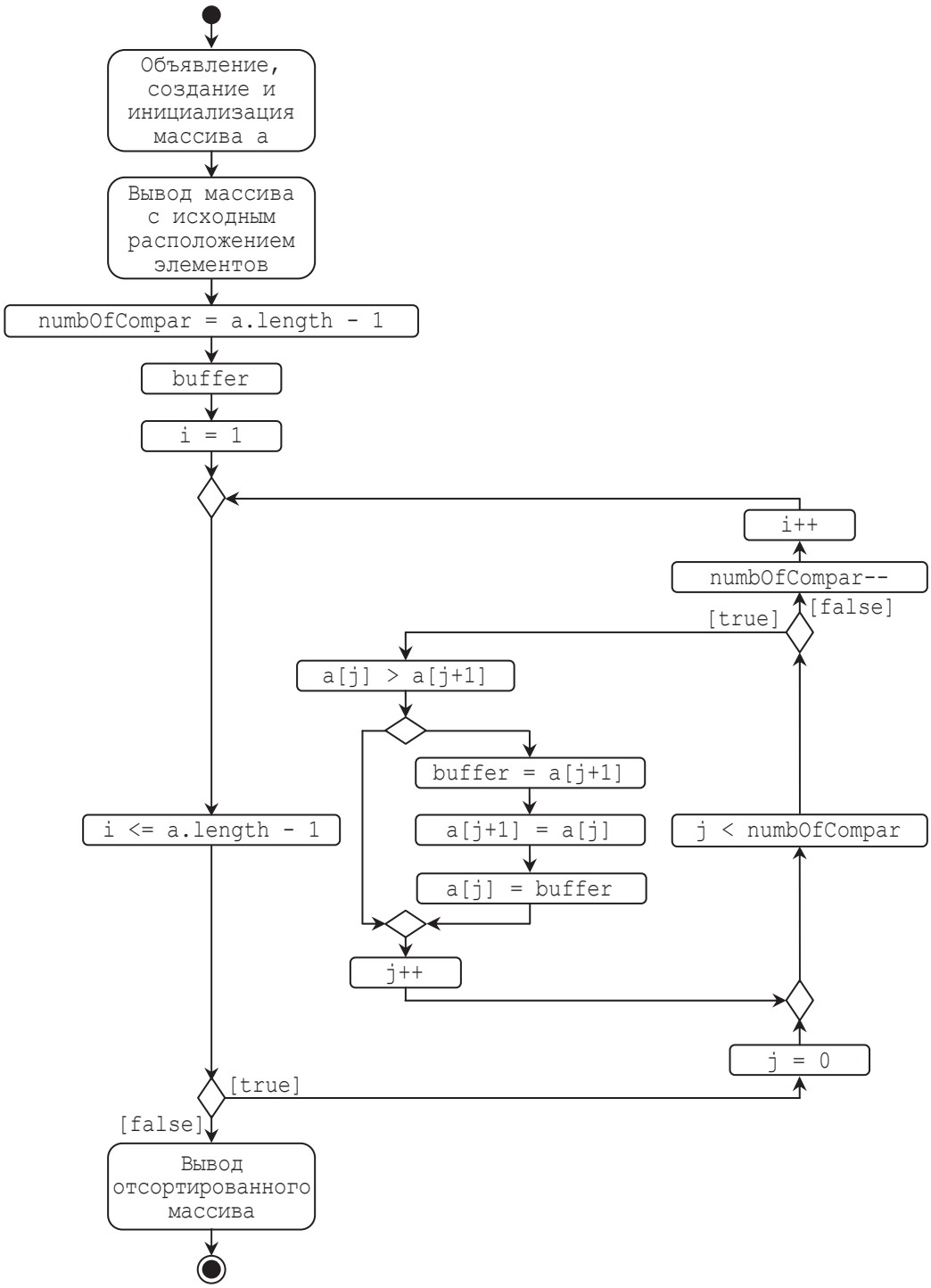



**Рис. 8.24.** Модель алгоритма сортировки одномерного массива в порядке возрастания значений элементов массива методом «пузырька»

внутреннего цикла является сравнение всех пар смежных элементов массива, начиная с первой пары, и перестановка элементов массива в паре, если предыдущий элемент массива оказывается больше последующего, поскольку массив сортируется в порядке возрастания значений своих элементов. Параметром внутреннего цикла является переменная `j`, которая инициализируется нулем (индексом первого элемента массива), Перед каждой последующей итерацией переменная `j` увеличивается на единицу. Условие окончания итераций определяется значением выражения `j < numbOfCompar`. Итерируемым блоком внутреннего цикла является блок развилка типа «одно условие — один блок». Условие ветвления этого блока записано в виде выражения `a[j] > a[j+1]` и определяет, является ли значение элемента массива с меньшим индексом больше, чем значение элемента массива с большим индексом. Если условие ветвления истинно, то выполняется блок в `true`-ветви, который реализует последовательность действий по перестановке элементов одномерного массива с использованием буферной ячейки (см. рис.8.17).

В заключительной части алгоритма расположен блок, осуществляющий вывод на экран монитора отсортированный массив.

Модель алгоритма, приведенная на рис. 8.24, позволяет отсортировать массив и в том случае, когда необходимо расположить элементы в порядке убывания их значений. Для этого необходимо сделать только одно изменение. В условии ветвления структуры развилка изменить символ «>» на символ «<» и записать условие ветвления в виде `[j] < a[j+1]`. На рис. 8.25 приведен код, в который отображается модель алгоритма, приведенная на рис. 8.24.

```
int[] a = {5, 4, 1, 3, 2};
// вывод a
int numbOfCompar = a.length – 1;
int buffer;
for(int i = 1; i <= a.length – 1; i++){
   for(int j = 0; j < numbOfCompar; j++){
     if(a[j] > a[j+1]){
        buffer = a[j+1];
        a[j+1] = a[j];
        a[j] = buffer;
        }
   }
   numbOfCompar--;
}
// вывод a
```

**Рис. 8.25.** Отображение модели алгоритма сортировки массива методов «пузырька» в код

Первое предложение кода на рис. 8.25 объявляет, создает и инициализирует массив с именем `a`, состоящий из пяти целых чисел. Как внешний, так и внутренний

циклы организованы на основе оператора `for`. Код в точности соответствует модели. В коде опущены предложения, кодирующие вывод элементов массива на экран монитора. Места, где должны быть расположены эти предложения, отмечены комментариями.

### 8.5.2. Сортировка методом прямого выбора

Сортировка методом прямого выбора, как и сортировка методом «пузырька», предполагает многократный просмотр массива. Рис. 8.26 иллюстрирует идею сортировки одномерного массива в порядке возрастания значений элементов методом прямого выбора. Для упрощения описания нумерация просмотров массива начинается с 0.

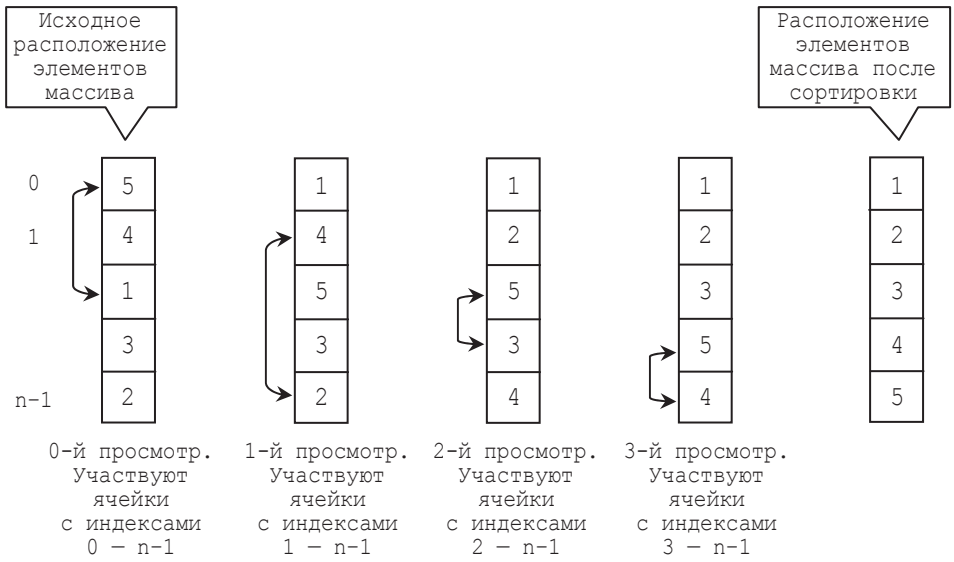



**Рис. 8.26.** Последовательность действий при сортировке одномерного массива методом прямого выбора. Стрелками отмечены элементы массива, подвергающиеся перестановке после завершения текущего просмотра

Просмотр с номером 0 начинается с ячейки с индексом 0. При просмотре с номером 0 находится индекс ячейки с элементом массива, имеющим наименьшее значение, после чего выполняется перестановка найденного элемента массива с элементов массива, находящихся в ячейке с индексом 0. Просмотр с номером 1 начинается с ячейки с индексом 1. При этом просмотре среди оставшихся ячеек находится ячейка с элементом массива, имеющим наименьшее значение, после чего выполняется перестановка найденного элемента массива с элементов массива, находящихся в ячейке с индексом 1. Таким образом, стартовой ячейкой на `i`-м просмотре является ячейка с индексом `i`, а найденный элемент с наименьшим значением

меняется местами с элементом в стартовой ячейке. Из сказанного можно заключить, что: (1) сортировка массива, состоящего из `n` элементов, требует `n-1` просмотров массива; (2) при просмотре с номером 0 количество просматриваемых элементов равно `n`, а при каждом последующем просмотре количество просматриваемых элементов уменьшается на единицу.

Трансформируем словесное описание сортировки массива методом прямого выбора в модель алгоритма. В основе структуры алгоритма сортировки методом прямого выбора, приведенной на рис. 8.27, как и в случае алгоритма сортировки массива методом пузырька, лежат вложенные циклы. Задачей внешнего цикла является управление просмотрами массива, а задачей внутреннего цикла — определение индекса ячейки, содержащей элемент с наименьшим значением, и перестановка содержимого этой ячейки с содержимым ячейки, индекс которой равен номеру просмотра.

Несколько структур типа блок в начальной части модели выполняют подготовительную работу, необходимую для сортировки. Вначале объявляется, создается, инициализируется и выводится на экран монитора массив, подвергающийся сортировке. Затем объявляются переменные: `minValue` (минимальное значение); `minIndex` (минимальный индекс) и `buffer` (буфер). Эти переменные используются во внутреннем цикле для следующих целей. После завершения текущего просмотра массива и выхода из внутреннего цикла в переменной `minValue` находится элемент массива с наименьшим значением, найденный в ходе просмотра, а в переменной `minIndex` — индекс этого элемента. Переменная с именем `buffer` используется для реализации алгоритма перестановки элементов массива с заданными индексами.

Как внешний, так и внутренний циклы относятся к типу «вначале проверка условия, затем итерация». Параметром внешнего цикла является переменная `i`, которая является счетчиком количества просмотров массива. Переменная `i` инициализируется 0 и на каждой итерации увеличивается на 1. Количество итераций внешнего цикла равно количеству просмотров массива, которое, в свою очередь, на 1 меньше количества элементов в массиве. Поэтому условие окончания итераций во внешнем цикле записано в виде выражения `i <= a.length - 1`. Завершение итераций во внешнем цикле означает завершение процесса сортировки. После выхода из внешнего цикла на экран монитора выводится отсортированный массив.

Внутренний цикл решает две задачи: (1) нахождение индекса элемента массива с наименьшим значением и (2) перестановка этого элемента с элементом массива, индекс которой равен номеру просмотра. Каждая новая итерация внутреннего цикла начинается с первого элемента в неотсортированной части массива. Поэтому параметр внутреннего цикла `j` инициализируется текущим значением переменной `i`. Следствием этого является то, что количество итераций во внутреннем цикле уменьшается на единицу при каждом новом просмотре массива. Тело внутреннего цикла реализует знакомый нам алгоритм нахождения первого наименьшего элемента массива и его индекс (см. рис. 8.15). Для этого используется структура развилка типа «одно условие — одно действие» с условием, записанным в виде выражения `a[j] < minValue`. После завершения итераций во внутреннем цикле и перед

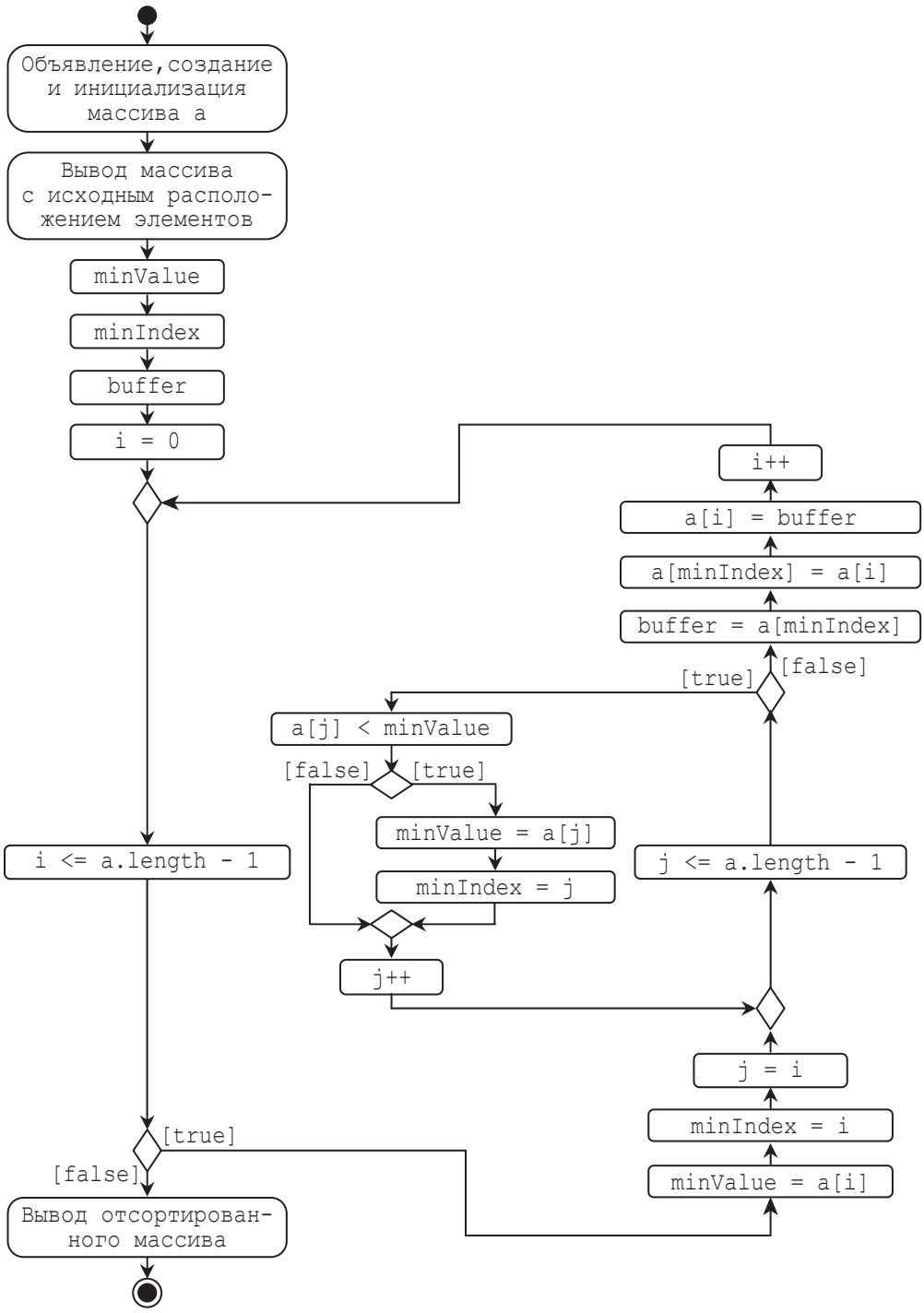



**Рис. 8.27.** Модель алгоритма сортировки одномерного массива в порядке возрастания значений элементов массива методом прямого выбора

началом очередной итерации во внешнем цикле реализуется алгоритм перестановки элементов массива с заданными индексами с использованием буферной ячейки (см. рис. 8.12).

На рис. 8.28 приведено отображение модели алгоритма сортировки массива методом прямого выбора в программный код.

```
int[] a = {5, 4, 1, 3, 2};
// вывод a
int minValue,
    minIndex,
    buffer;
for(int i = 0; i <= a.length — 1; i++){
   minValue = a[i];
   minIndex = i;
   for(int j = i; j <= a.length -1; j++){
      if(a[j] < minValue){
         minValue = a[j];
         minIndex = j;
      }
   }
   buffer = a[minIndex];
   a[minIndex] = a[i];
   a[i] = buffer;
}
// вывод a
```

**Рис. 8.28.** Отображение модели алгоритма сортировки массива методом прямого выбора в программный код

Код, приведенный на рис. 8.28, в точности соответствует модели. В качестве сортируемого массива используется тот же массив, состоящий из пяти целых чисел, который использовался при кодировании алгоритма сортировки методом «пузырька». В коде не записаны предложения, осуществляющие вывод массива на экран монитора. Места, где должны располагаться эти предложения, отмечены комментариями.

### 8.5.3. Сортировка методом вставки

Алгоритмы сортировки одномерных массивов методом «пузырька» и методом прямого выбора предполагают, что те элементы массива, которые в процессе сортировки попали в отсортированную часть, остаются на своих местах до окончания процесса сортировки. Алгоритм сортировки методом вставки работает иначе.

В процессе сортировки отсортированная часть массива все время изменяется, пополняясь новыми элементами.

Сортировка методом вставки учитывает уже имеющуюся упорядоченность массива и часто сравнивается с тем, каким образом игроки в карты упорядочивают свои карты во время игры. Если среди имеющихся в руках у игрока карт часть карт уже упорядочена, то вначале он извлекает из неупорядоченной части карт одну карту, затем в упорядоченной части определяет место, куда должна быть вставлена извлеченная карта, чтобы не нарушалась упорядоченность карт, и, наконец, извлеченная карта вставляется в это место. Процедура повторяется до тех пор, пока все карты не будут упорядочены. Рис. 8.29 иллюстрирует последовательность действий для одного из вариантов сортировки методом вставки массива, состоящего из пяти целых положительных чисел.

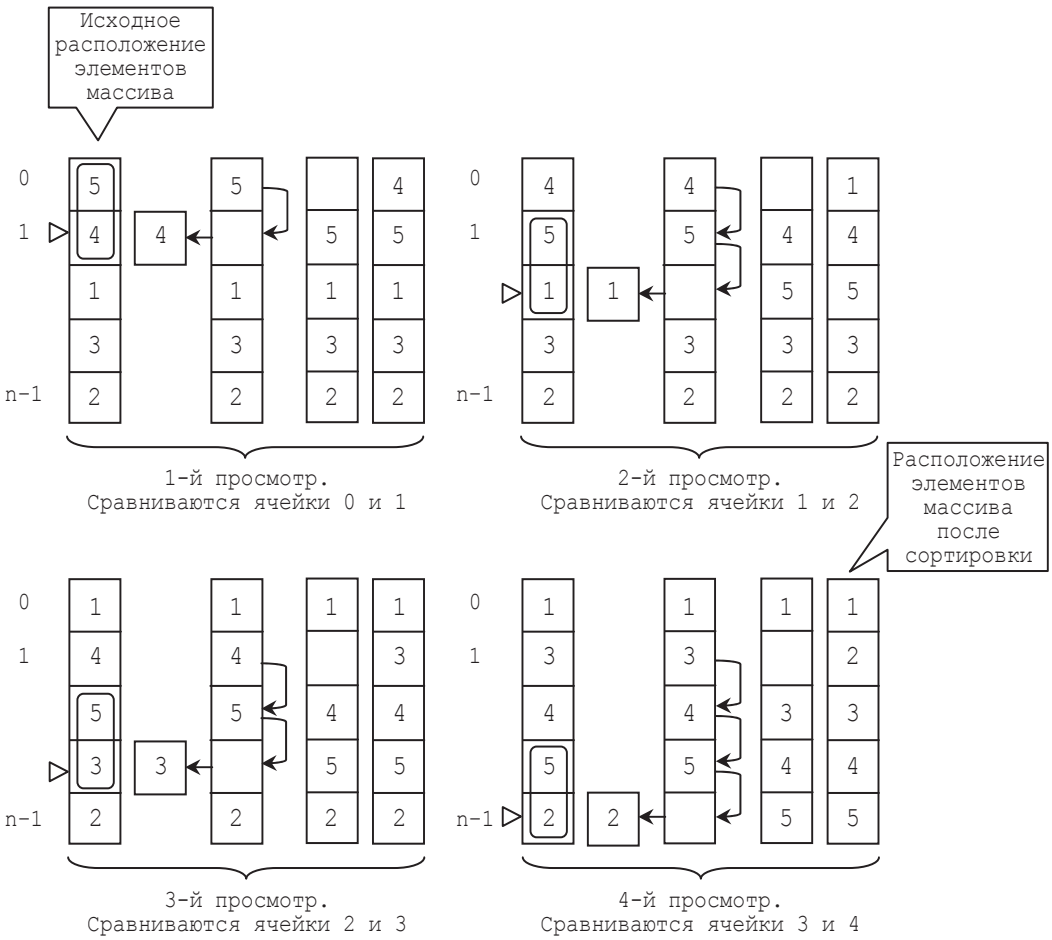



**Рис. 8.29.** Последовательность действий при сортировке одномерного массива методом вставки

Сортировка методом вставки предполагает многократный просмотр массива. Если массив состоит из `n` элементов, то сортировка завершается за `n-1` просмотр. После завершения первого просмотра в отсортированной части массива находится, по крайней мере, два элемента, а каждый последующий просмотр добавляет в отсортированную часть, по крайней мере, еще один элемент. На рис. 8.29 элементы массива в отсортированной части выделены жирным шрифтом.

При каждом просмотре массива последовательно выполняются четыре действия: (1) определяется элемент массива, подлежащий извлечению и последующей вставке в отсортированную часть массива; (2) извлекается найденный элемент массива из своей ячейки и запоминается в буферной ячейке; (3) сдвигаются элементы массива и освобождается одна из ячеек массива; (4) в освободившуюся ячейку массива записывается содержимое буферной ячейки.

В варианте сортировки массива, который иллюстрируется рис. 8.29, определение элемента массива, подлежащего извлечению и записи в буферную ячейку, выполняется следующим образом. Сравниваются элементы массива, находящиеся в смежных ячейках. Если массив сортируется в порядке возрастания значений элементов и в ячейке с большим индексом находится элемент массива с меньшим значением, то он переносится в буферную ячейку. В противном случае элемент массива остается в своей ячейке. При первом просмотре массива сравниваются элементы массива, находящиеся в ячейках с индексами 0 и 1, при втором — находящиеся в ячейках с индексами 1 и 2 и т. д.

На рис 8.29 сдвиг элементов массива осуществляется вниз, в сторону возрастания значений индексов ячеек. Предполагается, что *сдвиг элементов массива осуществляется последовательно, по одному элементу за один раз*. Последовательный сдвиг элементов массива продолжается до тех пор, пока значение очередного сдвигаемого элемента массива больше, чем значение элемента массива, находящегося в буферной ячейке.

Представим приведенное словесное описание метода сортировки массива методом вставки более формально в виде диаграммы деятельности, составленной из типовых поведенческих структур (см. рис. 7.26). На рис. 8.30 приведена модель алгоритма сортировки одномерного массива в порядке возрастания значений элементов в виде диаграммы деятельности.

В начальной части алгоритма выполняются подготовительные действия: объявляется, создается, инициализируется и выводится на экран монитора сортируемый массив с именем `a`; объявляются две целочисленные переменные с именами `buffer` (буфер) и `j`. Переменная `buffer` моделирует буферную ячейку, а переменная `j` является параметром внутреннего цикла. Отмеченные подготовительные действия моделируются последовательно расположенными структурами типа блок.

Центральная часть алгоритма состоит из вложенных циклов. Внешний цикл управляет просмотрами, а внутренний — действиями, выполняемыми при каждом просмотре. Оба цикла относятся к типу «вначале проверка условия, затем итерация».

Параметром внешнего цикла является переменная с именем `i`, которая изменяется в пределах от 1 до `a.length - 1` с шагом 1.

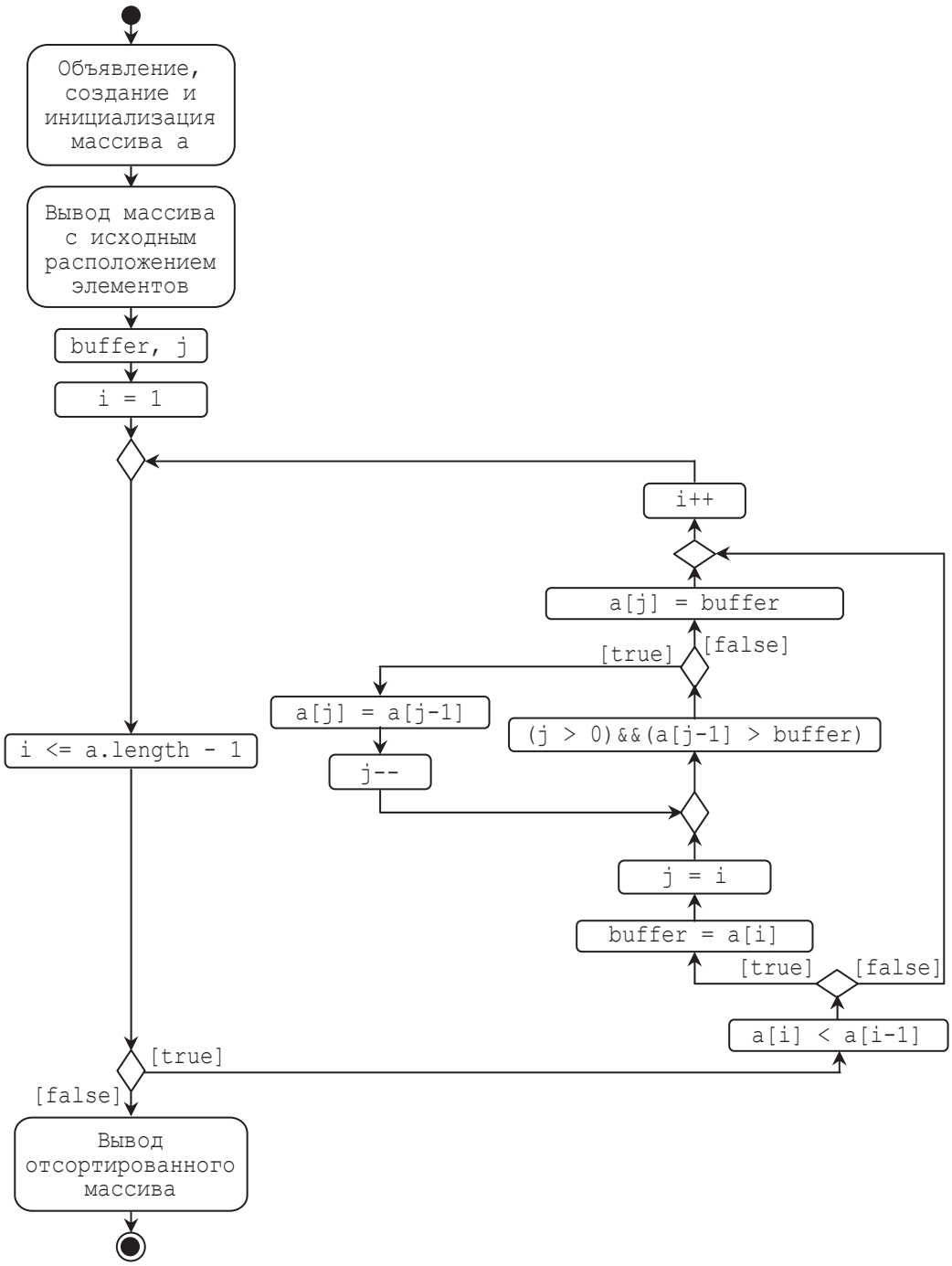



**Рис. 8.30.** Модель алгоритма сортировки одномерного массива в порядке возрастания значений элементов массива методом вставки

В начальной части итерируемого блока внешнего цикла располагается структура развилка типа «одно условие — один блок». Условие ветвления этой структуры записано в виде `a[i] < a[i-1]` и сравнивает два смежных элемента массива. Итерируемый блок внешнего цикла выполняется, если это условие принимает значение `true`. В этом случае в переменную с именем `buffer` записывается значение элемента массива `a[i]`, а затем выполняется внутренняя структура типа цикл. Целью внутреннего цикла является последовательный сдвиг элементов массива. Элементы массива сдвигаются до тех пор, пока значение очередного сдвигаемого элемента массива больше, чем значение переменной `buffer`, и/или не достигнута ячейка массива с индексом 0. Параметр внутреннего цикла `j` инициализируется текущим значением параметра внешнего цикла `i`, а условие завершения итераций записано в виде булевого выражения `(j > 0)&&(a[j-1] > buffer)`. Завершается итерируемый блок внешнего цикла блоком, который осуществляет перенос содержимого переменной `buffer` в ячейку массива, освободившуюся после сдвигов.

После выхода из внешнего цикла на экран монитора выводится отсортированный массив.

На рис. 8.31 приведен код, в который отображается модель, приведенная на рис. 8.30. Особенностью кода является то, что для организации внешнего и внутреннего цикла использованы различные операторы. Внешний цикл организован на основе оператора `for`, поскольку количество итераций внешнего цикла всегда одно и то же. Внутренний цикл организован на основе оператора `while-do`, поскольку количество итераций внутреннего цикла заранее не известно и определяется выражением `(j > 0)&&(a[j-1] > buffer)`. Количество итераций внутреннего цикла определяет количество сдвигов элементов массива, которое, в свою очередь, определяется тем, насколько упорядоченно размещены элементы исходного массива.

```
int[] a = {5, 4, 1, 3, 2};
// вывод a
int buffer, i, j;
for(i = 1; i <= a.length – 1; i++){
   if(a[j] < a[i-1]){
      buffer = a[i];
      j = i;
      while((j > 0)&&(a[j-1] > buffer)){
         a[j] = a[j-1];
         j--;
      }
      a[j] = buffer;
   }
}
// вывод a
```

**Рис. 8.31.** Отображение модели алгоритма сортировки массива методом вставки в программный код

Так же, как и в предыдущих примерах кодов, в коде, приведенном на рис. 8.31, не записаны предложения, кодирующие вывод массива на экран монитора.

## 8.6. Многомерные массивы

Массивы используются для хранения наборов однотипных данных и удобны тем, что обеспечивают прямой доступ к своим элементам. Прямой доступ означает, что доступ к значению элемента массива осуществляется путем указания значения индекса этого элемента. Внутренняя организация данных в одномерном массиве такова, что для получения доступа к элементу одномерного массива используется значение только одного индекса. Однако в некоторых случаях внутренняя организация данных одномерного массива неудобна и препятствует разработке надежных алгоритмов и программ. Покажем это на нескольких примерах.

На рис. 8.15 приведен алгоритм программы, оперирующий с одномерным массивом `temperature`, хранящим средние значения дневных температур в декабре. Массив состоит из 31 элемента, проиндексированных номерами дней месяца. Для того, чтобы получить доступ к значению средней дневной температуры 21 декабря, необходимо записать `temperature[21]`.

Представим, что мы используем одномерный массив для хранения средних дневных температур за весь прошлый год. В начальных ячейках массива располагаются значения температур за январь, в последующих — за февраль и т. д. Такой массив состоит из 365 элементов и обеспечивает прямой доступ к своим элементам. Для того, чтобы, например, получить доступ к значению средней дневной температуры 20 апреля прошлого года, нужно знать индекс элемента массива соответствующего 20 апреля прошлого года. Но чему он равен? Понятно, что для ответа на этот вопрос необходимо провести некоторые предварительные вычисления.

Представим теперь, что мы используем одномерный массив для хранения значений средних дневных температур за десять лет, начиная с 2000 года и заканчивая 2009 годом, и хотим получить доступ к значению средней дневной температуры 12 мая 2009 года. Это несложно сделать, если знать значение индекса соответствующего элемента массива. Однако его нужно уметь вычислить.

Внутренняя организация данных в многомерных массивах такова, что для получения прямого доступа к элементу массива необходимо указать значения нескольких индексов. Если мы используем массив для хранения средних дневных температур за весь прошлый год, то удобно использовать массив с двумя индексами. Пусть, например, первый индекс означает номер месяца, а второй — номер дня этого месяца. Тогда доступ к значению средней температуры 20 апреля прошлого года обеспечивается выражением `temperature[4][20]`. Если мы используем массив для хранения средних дневных температур за прошедшее десятилетие, то удобно использовать массив с тремя индексами. Первый индекс может соответствовать номеру года в десятилетии, второй — номеру месяца, а третий — номеру дня этого

месяца. Доступ к значению средней дневной температуры 12 мая 2009 года легко получить при помощи выражения `temperature[9][5][12]`.

Количество индексов, необходимых для обеспечения доступа к элементу массива, определяет размерность последнего. Для получения доступа к элементу двухмерного массива необходимо знать значения двух индексов, для получения доступа к элементу трехмерного массива — значения трех индексов и т. д.

В языке программирования Java многомерные массивы строятся из одномерных массивов. Если под словом «массив» понимать одномерный массив, то двухмерный массив можно рассматривать как «массив массивов», или массив, элементами которого являются массивы, а трехмерный массив — как «массив массивов массивов», или массив, элементами которого являются массивы, элементами которых, в свою очередь, являются массивы.

В практике императивного программирования часто используют двухмерные массивы. Если все одномерные массивы, из которых составляется двухмерный массив, имеют одинаковое количество элементов, то двухмерный массив удобно визуально представлять в виде матрицы, состоящей из строк и столбцов. Матрицы легко воспринимаются людьми и часто используются для представления и хранения наборов данных.

Рис. 8.32. иллюстрирует внутреннюю организацию двухмерного массива.

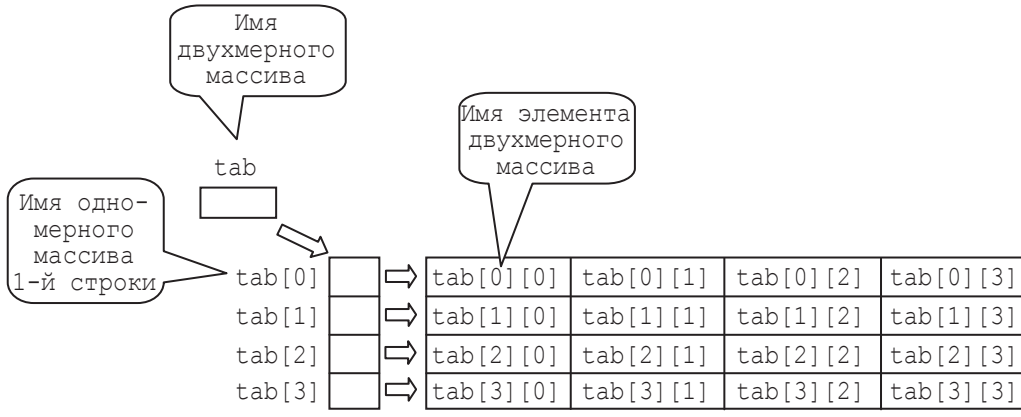



**Рис. 8.32.** Внутренняя организация двухмерного массива, состоящего из четырех строк и четырех столбцов

На рис. 8.32 имя двухмерного массива `tab` является ссылочной переменной, которая хранит ссылку на одномерный массив ссылок:

```
{tab[0], tab[1], tab[2], tab[3]}
```

Каждый элемент массива ссылок хранит ссылку на одномерный массив данных. Если количество элементов всех одномерных массивов, хранящих данные,

одинаково, то общее количество элементов массива tab равно произведению количества элементов в массиве ссылок на количество элементов в массиве данных. Для доступа к элементу массива tab необходимо знать значения двух индексов. Принято, что первый индекс является индексом строки, а второй — индексом столбца.

### 8.6.1. Объявление, создание и начальная инициализация многомерных массивов

Объявление многомерных массивов осуществляется при помощи предложений, имеющих такую же структуру, как и предложения, объявляющие одномерные массивы. Отличие заключается в количестве пар квадратных скобок. Если при объявлении одномерных массивов используется одна пара квадратных скобок, то при объявлении двухмерного массива — две пары, при объявлении трехмерного массива — три пары и т. д.

В языке программирования Java многомерные массивы, так же, как и одномерные, являются объектами соответствующих классов и имя массива это — ссылочная переменная, предназначенная для хранения ссылки на объект-массив. При объявлении многомерного массива детерминируется имя массива, тип элементов массива и размерность массива. На рис. 8.33 приведены примеры объявления многомерных массивов.

```
double[][] coefficientMatrix;   // объявлена ссылка на двухмерный массив
                                // с именем coefficientMatrix, состоящий
                                // из элементов типа double
float[][][] decadeTemperature;  // объявлена ссылка на трехмерный массив
                                // с именем decadeTemperature, состоящий
                                // из элементов типа float
Employee[][] researchers;       // объявлена ссылка на двухмерный массив
                                // с именем researchers, состоящий
                                // из ссылок на объекты класса Employee
```

**Рис. 8.33.** Примеры объявления многомерных массивов

В предложениях на рис. 8.33 двойные квадратные скобки, при помощи которых указывается размерность массива, записаны после имени типа элементов массива, однако, допустима нотация, когда эти скобки записываются после имени массива. Поэтому примеры объявления массивов, приведенные на рис. 8.33, могут быть записаны следующим образом:

```
double coefficientMatrix[][];
float decadeTemperature[][][];
Employee researchers[][];
```

Выполнение любого из предложений, приведенных на рис. 8.33, размещает в памяти компьютера не сам массив, а только переменную ссылочного типа, которая после создания массива как объекта соответствующего класса будет содержать значение ссылки на этот объект-массив.

Создание объекта многомерного массива выполняется при помощи предложения со служебным словом `new`, которое мы ранее использовали для создания объекта одномерного массива. Если массив уже объявлен, то в этом предложении детерминируется количество элементов массива. В связи с тем, что в языке программирования Java многомерные массивы «собираются» из одномерных, имеется несколько способов создания многомерных массивов. Рассмотрим их на примере двухмерного массива.

Первый способ создания двухмерного массива предназначен для использования в тех случаях, когда массив имеет *одинаковое количество элементов в каждой строке*, или, когда все одномерные массивы, из которых составляются строки двухмерного массива, имеют одинаковое количество элементов. Именно такой массив является моделью плоской таблицы или матрицы.

Если массив уже объявлен, то предложение, при помощи которого кодируется создание массива с одинаковым количеством элементов в каждой строке, имеет следующую структуру:

```
<имя массива> = new <тип элементов массива> [<количество строк>]
                    [<количество столбцов>];
```

Например, создание двухмерного массива с именем `coefficientMatrix` (матрица коэффициентов), состоящего из элементов типа `double` и объявленного на рис. 8.33, можно выполнить при помощи предложения

```
coefficientMatrix = new double [6][7];
```

Выполнение приведенного предложения порождает в памяти массив, состоящий из 6 строк по 7 элементов в каждой строке, значения элементов которого проинициализированы нулями в соответствии с правилом умолчания.

В одном предложении можно совместить и объявление, и создание двухмерного массива. Если бы массив с именем `coefficientMatrix` ранее не был объявлен, то его можно и объявить, и создать при помощи предложения.

```
double [][] coefficientMatrix = new double [6][7];
```

Второй способ создания двухмерных массивов используется в тех случаях, когда массив должен иметь *различное количество элементов в каждой строке*, что соответствует случаю, когда одномерные массивы, из которых составляются строки двухмерного массива, имеют различное количество элементов. Создание массива выполняется в два этапа. На первом этапе при помощи одного предложения создается массив ссылок на массивы-строки, а на втором — при помощи нескольких

предложений создаются сами массивы-строки. Поскольку массивы-строки создаются независимыми предложениями, то имеется возможность определять различное количество элементов в каждой строке. Если применить рассматриваемый способ создания двухмерного массива к массиву `tab`, изображенному на рис. 8.32, то на первом этапе создается массив `{tab[0], tab[1], tab[2], tab[3]}`, представляющий собой ссылки на строки-массивы, а на втором этапе — сами строки-массивы. Фрагмент кода, реализующий второй из рассматриваемых способов создания двухмерного массива, приведен на рис. 8.34.

```
int[][] tab;
tab = new int[4][];
tab[0] = new int[4];
tab[1] = new int[3];
tab[2] = new int[2];
tab[3] = new int[1];
```

**Рис. 8.34.** Объявление и создание двухмерного массива с различным количеством элементов в строках

Первое предложение на рис. 8.34 объявляет ссылочную переменную с именем `tab`. Второе предложение создает одномерный массив, состоящий из четырех элементов, предназначенных для хранения ссылок на строки массива `tab`. Если на этом этапе определить количество элементов массива при помощи выражения `tab.length`, то получим число 4.

Предложения с третьего по шестое создают объекты класса одномерных массивов, представляющие собой строки массива `tab`. Первая строка-массив состоит из четырех элементов, а каждая последующая содержит на один элемент меньше. Количество элементов в каждой из строк-массивов сохраняется в поле `length` соответствующего объекта-массива. Поэтому выражение `tab[2].length` возвращает число 2.

На рис. 8.35 изображена организация двухмерного массива, созданного в результате выполнения кода на рис. 8.34.

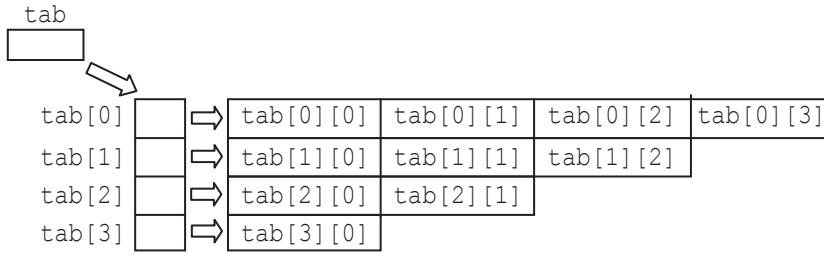



**Рис. 8.35.** Внутренняя организация двухмерного массива, созданного в результате выполнения кода на рис. 8.34

Для получения доступа к элементу двухмерного массива необходимо знать его индексы. Напомним, что значения индексов массивов начинаются с нуля, и для случая двухмерного массива первый индекс интерпретируется как номер строки, а второй — как номер столбца. Доступ к элементу массива, созданному при помощи кода, приведенного на рис. 8.34 и находящемуся в первой строке и первом столбце, можно получить при помощи его имени в виде `tab[0][0]`.

В некоторых случаях инициализация двухмерного массива может выполняться при помощи набора предложений с операторами присваивания. Например,

```
tabe[0][0] = 24;
tabe[0][1] = 12;
tabe[0][2] = 7;
tabe[0][3] = 19;
tabe[1][0] = 10;
tabe[1][1] = 5;
      .  .  .
```

Двухмерный массив, так же, как и одномерный, можно объявить, создать и проинициализировать при помощи одного предложения со списком значений элементов массива в фигурных скобках. Например:

```
int[][] tab = {{24,12,7,19},{10,5,1},{22,11},{3}};
```

Если двухмерный массив создается и инициализируется при помощи предложения, содержащего список значений его элементов в фигурных скобках, то в квадратных скобках не указывается количество строк и столбцов, а в правой части предложения последовательно и построчно в фигурных скобках размещаются списки значений элементов массива. Количество строк, а также количество элементов в каждой строке задаются опосредовано списками в фигурных скобках. В приведенном примере двухмерный массив `table` состоит из четырех строк разной длины. Элементы первой строки проинициализированы числами: 24, 12, 7, 19; элементы второй строки — числами: 10, 5, 1 и т. д.

Если есть правило, связывающее значения индексов со значениями элементов массива, то для инициализации двухмерных массивов используются вложенные циклы. Переменная-параметр внешнего цикла принимает значения первого индекса, а переменная-параметр внутреннего цикла принимает значения второго индекса и на каждой итерации внешнего цикла просматривает элементы текущего массива-строки.

Проиллюстрируем использование вложенных циклов для инициализации элементов двухмерного массива на примере решения следующей задачи. Необходимо создать двухмерный массив, состоящий из целых чисел и моделирующий квадратную матрицу, состоящую из 10 строк и 10 столбцов, и затем проинициализировать элементы этого массива значениями в соответствии с рис. 8.36.

Квадратные матрицы часто используются для представления данных при решении научно-технических и экономических задач. При описании матриц и алгоритмов работы с ними используется специальная терминология.

Главной диагональю квадратной матрицы называются отрезок, соединяющий верхний левый и нижний правый углы матрицы.

Побочной диагональю квадратной матрицы называются отрезок, соединяющий верхний правый и нижний левый углы матрицы.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **1** |
| 0 | **2** | 0 | 0 | 0 | 0 | 0 | 0 | **2** | 0 |
| 0 | 0 | **3** | 0 | 0 | 0 | 0 | **3** | 0 | 0 |
| 0 | 0 | 0 | **4** | 0 | 0 | **4** | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | **5** | **5** | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | **6** | **6** | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | **7** | 0 | 0 | **7** | 0 | 0 | 0 |
| 0 | 0 | **8** | 0 | 0 | 0 | 0 | **8** | 0 | 0 |
| 0 | **9** | 0 | 0 | 0 | 0 | 0 | 0 | **9** | 0 |
| **10** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **10** |

**Рис. 8.36.** Начальная инициализация квадратной матрицы

Если матрица состоит из `n` строк и `n` столбцов, индекс текущей строки представляется значением переменной `i`, а индекс текущего столбца — переменной `j`, то для элементов матрицы имеют место следующие соотношения между индексами.

Если элемент матрицы расположен на главной диагонали, то для такого элемента значение индекса строки совпадает со значением индекса столбца: `i=j`.

Если элемент матрицы расположен выше главной диагонали, то для такого элемента значение индекса строки меньше, чем значение индекса столбца: `i<j`.

Если элемент матрицы расположен ниже главной диагонали, то для такого элемента значение индекса строки больше, чем значение индекса столбца: `i>j`.

Если элемент матрицы расположен на побочной диагонали, то для такого элемента справедливо равенство: `i+j = n-1`.

Если элемент матрицы расположен выше побочной диагонали, то для такого элемента справедливо неравенство: `i+j < n-1`.

Если элемент матрицы расположен ниже побочной диагонали, то для него справедливо неравенство: `i+j > n-1`.

Рис. 8.37 иллюстрирует соотношения между индексами элементов квадратной матрицы в зависимости от их расположения по отношению к главной или побочной диагонали.

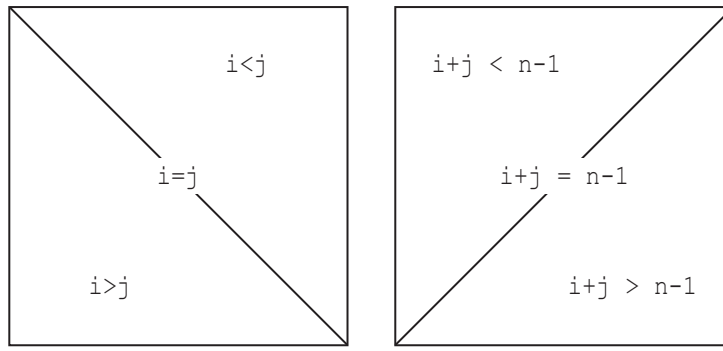



**Рис. 8.37.** Соотношения между индексами элементов квадратной матрицы в зависимости от их расположения по отношению к главной или побочной диагонали

В требуемой начальной инициализации элементов квадратной матрицы, изображенной на рис. 8.36, легко заметить правило, связывающее значение элемента матрицы со значениями его индексов. Это правило может быть сформулировано следующим образом. Если элемент находится на главной или побочной диагонали, то его значение на единицу больше, чем значение индекса строки, а если элемент не находится на главной или побочной диагонали, то его значение равно нулю.

Модель алгоритма, решающего сформулированную задачу начальной инициализации квадратной матрицы, приведена на рис. 8.38.

В начальной части алгоритма при помощи структуры типа блок объявляется и создается двухмерный массив с именем `matrix` , подлежащий инициализации.

Центральная часть алгоритма представляет собой вложенную структуру типа цикл, у которой и внешний, и внутренний циклы относятся к типу «вначале проверка условия, затем итерация». Параметром внешнего цикла является переменная-индекс строки с именем `i`. При помощи внешнего цикла осуществляется просмотр всех строк массива. Поэтому параметр внешнего цикла инициализируется значением 0 (индекс первой строки), по завершению каждой итерации увеличивается на 1, а условием окончания итераций является завершение просмотра последней строки, записанное в виде выражения `i < 10`.

Внутренний цикл просматривает все элементы массива в пределах текущей строки. Параметром внутреннего цикла является переменная-индекс столбца с именем `j`, который инициализируется значением 0 (индекс первого столбца), модифицируется путем увеличения на 1, а условием окончания итераций внутреннего цикла является достижение конца строки. Это условие записано в виде выражения `j < 10`. Телом внутреннего цикла является структура развилка типа «одно условие — два блока». Условием ветвления этой структуры типа развилка является принадлежность элемента массива главной или побочной диагонали. Это условие

записано в виде выражения `(i==j)||(i+j=9)`. Если приведенное выражение принимает значение `true`, (элемент находится на главной или побочной диагонали), то выполняется блок, который присваивает этому элементу массива значение индекса текущей строки, увеличенное на 1. В противном случае выполняется блок, который присваивает элементу массива значение, равное 0.

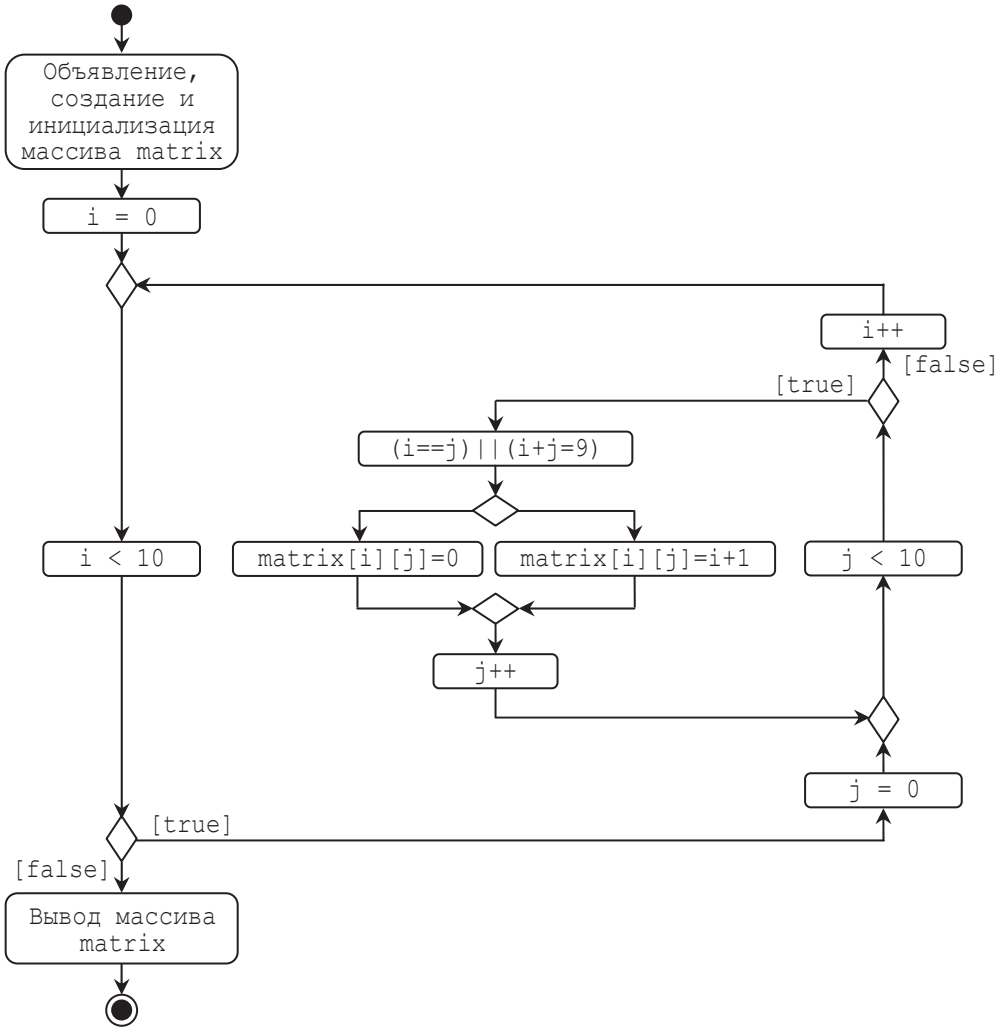



**Рис. 8.38.** Модель алгоритма инициализации двухмерного массива при помощи вложенных циклов

В заключительной части алгоритма размещена структура типа блок, при помощи которой на экран монитора выводится результат инициализации массива.

На рис. 8.39 приведен фрагмент кода, соответствующий модели, приведенной на рис. 8.38.

```
int[][] matrix = new int[10][10];
for(int i = 0; i < 10; i++)
   for(int j = 0; j < 10; j++)
      if((i==j)||(i+j=9))
         matrix[i][j]=i+1;
      else
         matrix[i][j]=0;
// вывод matrix
```

**Рис. 8.39.** Отображение модели алгоритма инициализации двухмерного массива, приведенного на рис. 8.38, в программный код

Код, приведенный на рис. 8.39, в точности соответствует модели. В нем не приведены предложения, осуществляющие вывод массива `matrix` на экран монитора. Это место отмечено комментарием. Код можно упростить. После выполнения первого предложения в памяти компьютера будет размещен массив, проинициализированный значениями в соответствии с правилом умолчания. Поскольку массив состоит из чисел типа `int`, то по правилу умолчания он будет проинициализирован нулевыми значениями. Поэтому в теле внутреннего цикла можно использовать оператор `if` типа «одно условие — один блок» и исключить из кода строки

```
else
   matrix[i][j]=0;
```

Отметим, однако, что вариант кода на рис. 8.39 предпочтительнее. Во-первых, при анализе кода не возникает вопрос о том, что происходит с элементами массива, не находящихся на главной и побочной диагоналях, а во-вторых, код легко модифицировать и заполнять элементы массива, не расположенные на диагоналях любыми числами.

### 8.6.2. Использование оператора for-each для работы с многомерными массивами

Оператор `for-each` применим для работы с многомерными массивами. Рассмотрим особенности применения оператора `for-each` при работе с многомерными массивами на примере двухмерного массива.

Как было отмечено ранее (см. подраздел 8.3), оператор `for-each` самостоятельно организует итерационный процесс обработки элементов массива. Программист не объявляет, не инициализирует, не модифицирует параметр/параметры цикла и не формулирует условие окончания итераций. Все эти действия обеспечиваются самим оператором, исходя из предположения, что в итерационном процессе необходимо просмотреть все элементы массива. Оператор `for-each` на каждой итерации

копирует очередной элемент массива в буферную переменную, доступную программисту. Такая степень «автоматизации» итерационного процесса исключает логическую ошибку, приводящую на этапе выполнения кода к исключительной ситуации типа «выход за пределы массива».

При использовании оператора `for-each` для работы с двухмерным массивом необходимо, как и при использовании обычного оператора `for` либо оператора `while`, организовать вложенные циклы. В процессе выполнения итераций внешнего цикла оператор `for-each` копирует в буфер текущую строку двухмерного массива, а в процессе выполнения итераций внутреннего цикла оператор `for-each` копирует в буфер очередной элемента текущей строки. Таким образом, буфером внешнего оператора `for-each` является одномерный массив, а буфером внутреннего оператора `for-each` — обычная переменная.

Буфер оператора `for-each` объявляется в его заголовке и называется итерационной переменной. Ясно, что поскольку в буфер как внешнего, так и внутреннего операторов `for-each` копируются данные из массива, то типы итерационных переменных должны либо совпадать, либо быть совместимыми, с типом элементов массива.

Проиллюстрируем отмеченную специфику использования оператора `for-each` для работы с двухмерным массивом на примере кодирования следующей задачи. Пусть необходимо найти сумму всех элементов двухмерного массива, созданного и проинициализированного при помощи кода, приведенного на рис. 8.39. На рис. 8.40 приведен фрагмент кода, решающий эту задачу.

```
int[][] matrix = new int[10][10];
// инициализация элементов массива
int sum = 0;
for(int row[] : matrix)
   for(int x : row)
      sum += x;
// вывод sum
```

**Рис. 8.40.** Пример кода, использующего оператор `for-each` для работы с двухмерным массивом

Внешний цикл организован на основе оператора `for-each`. В заголовке этого цикла объявлена итерационная переменная с именем `row` (строка) и указано имя массива `matrix`, элементы которого участвуют в итерационном процессе. Итерационная переменная `row` объявлена как одномерный массив элементов типа `int` и является буфером, в который на каждой итерации внешнего цикла копируется текущая строка массива `matrix`. Поскольку массив `matrix` уже объявлен и количество его строк и столбцов известно, то нет необходимости указывать количество элементов массива `row`.

Внутренний цикл также организован на основе оператора `for-each`. В заголовке внутреннего цикла объявлена обычная итерационная переменная с именем `x` и

указано имя массива `row`, элементы которого участвуют в итерационном процессе внутреннего цикла. Телом внутреннего цикла является предложение `sum += x`, при помощи которого к ранее накопленной сумме элементов массива `matrix` прибавляется очередной элемент.

### 8.6.3. Базовые алгоритмы работы с двухмерными массивами

Понятие «базовые алгоритмы» работы с массивами не точное и предполагает список тех, относительно простых, алгоритмов, которые чаще других используются при построении более сложных и изощренных алгоритмов работы с массивами. Список базовых алгоритмов работы с двухмерным массивом примерно такой же, как и список базовых алгоритмов работы с одномерными массивами:

- нахождение суммы/произведения элементов массива, состоящего из числовых данных;
- поиска первого/последнего элемента массива с заданным значением.
- поиск индексов элемента массива с наибольшим/наименьшим значением;
- перестановка строк или столбцов массива с заданными индексами;

Центральной частью всех базовых алгоритмов работы с двухмерными массивами являются вложенные циклы, в которых и внешний, и внутренний циклы относятся к типу «вначале проверка условия, затем итерация». Внешний цикл оперирует со строками (или с индексами строк), а внутренний — с элементами строки (или индексами столбцов).

В качестве примера базового алгоритма работы с двухмерным массивом рассмотрим алгоритм поиска элемента с н аибольшим значением и индексов этого элемента.

Сформулируем задачу следующим образом. Пусть имеется двухмерный массив, предназначенный для хранения средних дневных температур в 2013 году. Массив состоит из 12 строк. Строки соответствуют месяцам года. Строка с индексом 0 состоит из 31 элемента и хранит средние дневные температуры января, строка с индексом 1 состоит из 28 элементов и хранит средние дневные температуры февраля и т. д. Необходимо найти наибольшее зарегистрированное значение дневной температуры в 2013 году и определить месяц и день, когда эта температура была зарегистрирована. Результат поиска вывести на экран монитора в виде легко воспринимаемого предложения, например:

```
Наибольшая средняя дневная температура: 32°C
зарегистрирована 16 августа.
```

Как следует из формулировки задачи, массив, хранящий значения температур, состоит из не одинакового количества элементов в строках. Индекс строки соответствует номеру месяца, а индекс столбца — номеру дня недели. Поскольку нумерация строк и столбцов в двухмерном массиве начинается с нуля, то для определения

номера месяца и номера дня в общепринятом значении необходимо значения индексов строки и столбца увеличить на единицу. Алгоритм должен предусматривать трансформацию индекса строки в наименование месяца.

Решение описанной задачи можно разбить на три части: (1) объявление, создание и инициализация двухмерного массива, состоящего из различного количества элементов в строках; (2) поиска элемента с наибольшим значением и индексов строки и столбца для этого элемента; (3) трансформацию найденного индекса строки в наименование месяца и вывод результата на экран монитора.

На рис. 8.41 приведена модель алгоритма решения сформулированной задачи.

В начальной части алгоритма расположена цепочка структур типа блок, которые выполняют следующие действия: (1) объявление и создание двухмерного массива с именем `temp2013`, предназначенного для хранения средних значений дневных температур в 2013 году; (2) инициализация массива `temp2013` начальными значениями; (2) объявление и инициализация переменной с именем `maxT` (максимальная температура), которая после завершения работы алгоритма должна содержать искомое значение температуры; (3) объявление и инициализация двух переменных с именами `monthIndex` (индекс месяца) и `dayIndex` (индекс дня), которые после завершения работы алгоритма должны содержать индексы строки и столбца элемента массива с наибольшим значением температуры; (4) объявление переменной с именем `monthName`, предназначенной для хранения имени месяца, в котором была зарегистрирована наибольшая температура.

Центральная часть алгоритма представляет собой вложенные циклы, составленные из двух циклов типа «вначале проверка условия, затем итерация». Параметром внешнего цикла является переменная `monthCounter` (счетчик месяцев). Начальная инициализация, модификация, а также условие окончания этого параметра таковы, что на каждой итерации внешнего цикла осуществляется доступ к строке `temp2013[month]`, хранящей значения температур текущего месяца. Параметром внутреннего цикла является переменная `dayCounter` (счетчик дней). Начальная инициализация, модификация, а также условие окончания этого параметра таковы, что на каждой итерации внутреннего цикла осуществляется доступ к очередному элементу массива `temp2013[month][day]`, находящемуся в текущей строке `temp2013[month]`. Тело внутреннего цикла состоит из структуры развилка типа «одно условие — один блок». Задачей этой структуры развилка является определение того, является ли значение текущего значения температуры `temp2013[month][day]` больше, чем предыдущее наибольшее значение температуры в переменной `maxT`. Если текущая температура больше, чем предыдущая наибольшая температура, то выполняются три действия: (1) текущая температура заменяет предыдущее наибольшее значение (`maxT = temp2013[month][day]`); (2) индекс строки (номер месяца), содержащей элемент массива с наибольшей температурой, заменяет предыдущий номер месяца (`monthIndex = monthCounter`); (3) индекс столбца (номер дня), содержащий элемент массива с наибольшей температурой, заменяет предыдущий номер дня (`dayIndex = dayCounter`).

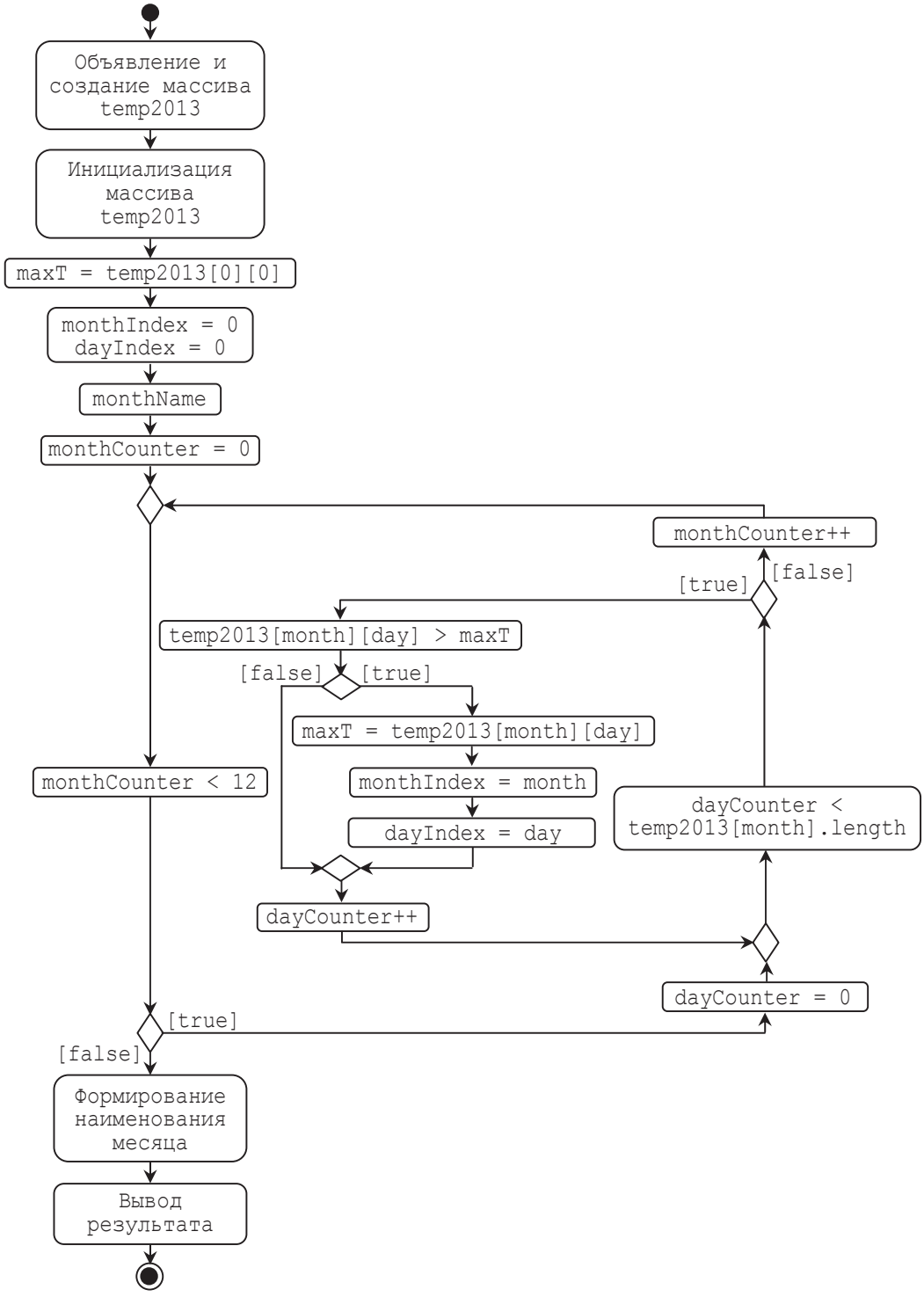



**Рис. 8.41.** Модель алгоритма поиска элемента с наибольшим значением в двухмерном массиве

Во внешнем цикле выполняется 12 итераций, а количество итераций внутреннего цикла равно количеству дней в текущем месяце и определяется значением поля `length` при помощи выражения `temp2013[month].length`.

После завершения всех итераций и выхода из внешнего цикла осуществляется трансформация номера месяца в его наименование и вывод результатов работы алгоритма на экран монитора в форме, указанной в формулировке задачи. Эти действия выполняются цепочкой из двух структур типа блок, расположенных в заключительной части алгоритма.

На рис. 8.42 приведен фрагмент кода модели алгоритма, изображенной на рис. 8.41.

Первое предложение объявляет ссылочную переменную с именем `temp2013`, предназначенную для хранения ссылки на двухмерный массив, состоящий из числовых данных типа `double`. Второе предложение кода создает одномерный массив, состоящий из 12 элементов, предназначенных для хранения ссылок на строки массива `temp2013`. Поскольку отдельные строки массива состоят из различного количества элементов, то последующие двенадцать предложений создают эти строки. Например, предложение

```
temp2013[0] = new double[31];
```

создает первую строку, которая состоит из 31 элемента и предназначена для хранения значений средних дневных температур в январе.

После объявления и создания массива в его элементы будут автоматически записаны нулевые значения, в соответствии с правилом умолчания. Для того, чтобы массив был заполнен средними дневными температурами, необходима начальная инициализация. Предполагается, что начальная инициализация осуществляется путем ввода значений температур с клавиатуры. Предложения, кодирующие начальную инициализацию, не приведены. Их место в коде отмечено комментарием.

В центральной части кода записаны предложения, кодирующие вложенные циклы, при помощи которых осуществляется поиск наибольшего значения дневной температуры и определение месяца и дня, когда эта температура была зарегистрирована. Для организации внешнего и внутреннего циклов используются операторы `for`.

Для выполнения необходимых действий в итерационных процессах, помимо параметров циклов `monthCounter` и `dayCounter`, необходимы следующие переменные: `maxT` (текущая максимальная температура); `monthIndex` (индекс месяца, в котором была зарегестрирована текущая максимальная температура); `dayIndex` (индекс дня, в котором была зарегестрирована текущая максимальная температура); `monthName` (наименование месяца, в котором была зарегестрирована текущая максимальная температура). Предложения, записанные непосредственно после инициализации массива, объявляют и инициализируют перечисленные переменные. Согласно принятой организации массива `temp2013`, индексу месяца `monthIndex` соответствует индекс строки, а индексу дня `dayIndex` — индекс элемента в строке.

```
double[][] temp2013;
temp2013 = new double [12][];
temp2013[0]  = new double[31]; //январь
temp2013[1]  = new double[28]; //февраль
temp2013[2]  = new double[31]; //март
temp2013[3]  = new double[30]; //апрель
temp2013[4]  = new double[31]; //май
temp2013[5]  = new double[30]; //июнь
temp2013[6]  = new double[31]; //июль
temp2013[7]  = new double[31]; //август
temp2013[8]  = new double[30]; //сентябрь
temp2013[9]  = new double[31]; //октябрь
temp2013[10] = new double[30]; //ноябрь
temp2013[11] = new double[31]; //декабрь
//инициализация массива temp2013 путем ввода значений с клавиатуры
double maxT = temp2013[0][0];
int monthIndex = 0,
    dayIndex = 0;
String monthName = null;
for(int monthCounter = 0; monthCounter < 12; monthCounter++)
   for(int dayCounter = 0; dayCounter < temp2013[month].length; dayCounter++){
      if(temp2013[month][day] > maxT){
         maxT = temp2013[month][day];
         monthIndex = monthCounter;
         dayIndex = dayCounter;
         }
   }
switch(monthIndex){
   case 0: monthName = " января";
           break;
   case 1: monthName = " февраля";
           break;
   case 2: monthName = " марта";
           break;
   case 3: monthName = " апреля";
           break;
   case 4: monthName = " мая";
           break;
   case 5: monthName = " июня";
           break;
   case 6: monthName = " июля";
           break;
   case 7: monthName = " августа";
           break;
   case 8: monthName = " сентября";
           break;
   case 9: monthName = " октября";
           break;
   case 10: monthName = " ноября";
            break;
   case 11: monthName = " декабря";
            break;
}
System.out.println("Наибольшая средняя дневная температура: " + maxT + "°C");
System.out.println("зарегистрирована " + (dayIndex + 1) + monthName);
```

**Рис. 8.42.** Отображение алгоритма поиска элемента с наибольшим значением в двухмерном массиве в программный код

В заголовке внешнего цикла условие окончания итераций записано в виде выражения `monthCounter < 12`, поскольку количество строк массива неизменно и равно 12. Количество элементов в текущей строке является переменной величиной, поэтому в выражении, определяющем условие окончания итераций во внутреннем цикле, использовано значение поля `length`, а условие записано в виде `dayCounter < temp2013[month].length`.

Действия, связанные с поиском наибольшей дневной температуры и определения месяца и дня, когда эта температура была зарегистрирована, кодируются в теле внутреннего цикла при помощи оператора `if`. Условие ветвления этого оператора записано в виде выражения `temp2013[month][day] > maxT`, при помощи которого на каждой итерации внутреннего цикла проверяется, не превышает ли температура в текущем элементе массива `temp2013[month][day]` предыдущее максимальное значение температуры, находящееся в `maxT`.

Если текущая температура больше, чем значение температуры в `maxT`, то последовательно выполняются три предложения

```
maxT = temp2013[month][day];
monthIndex = monthCounter;
dayIndex = dayCounter;
```

при помощи которых обновляются значения переменных `maxT`, `monthIndex` и `dayIndex`.

После выхода из внешнего цикла, при помощи оператора `switch` осуществляется преобразование значения индекса месяца, в котором было зарегестрировано наибольшее значение температуры (переменная `monthIndex`), в наименование месяца в родительном падеже, чтобы можно было вывести на экран монитора корректное предложение. В операторе `switch` после каждой `case`-структуры записан оператор `break`, прерывающий работу оператора `switch`, если значение переменной `monthIndex` равно значению литерала в этой `case`-структуре.

Результаты поиска выводятся на экран в виде связного предложения, разделенного на две строки, в котором указывается найденное значение температуры, а также наименование месяца и день, когда эта температура была зарегистрирована.

### 8.6.4. Сортировка двухмерных массивов

Данные в двухмерном массиве изначально частично упорядочены, поскольку сгруппированы в строки и столбцы, которые расположены в порядке возрастания значений их индексов. Поэтому упорядочивание данных в двухмерном массиве путем их сортировки должно учитывать отмеченную изначальную упорядоченность. Существует несколько логических возможностей определения понятия сортировки двухмерных массивов, однако, как правило, под сортировкой двухмерного массива понимают либо сортировку элементов заданной строки путем перестановки

столбцов, либо сортировку элементов заданного столбца путем перестановки строк. Поясним это на следующем примере. Пусть пять человек в течение шести месяцев (с января по июнь включительно) работали над некоторым проектом и ежемесячно получали заработную плату, величина которой определялась выполненной работой. Сведения о заработной плате всех участников проекта, полученной за время работы над проектом, можно представить в виде двухмерного массива, состоящего из пяти строк и шести столбцов, в котором каждая строка содержит сведения о заработной плате одного человека. Например, первая строка содержит сведения о заработной плате Иванова, вторая — сведения о заработной плате Петрова, и т. д. В этом случае каждый столбец содержит сведения о заработной плате всех участников проекта в конкретном месяце. Например, первый столбец содержит сведения о заработной плате за январь, второй — за февраль и т. д.

Представим себе, что нам необходимо расположить участников проекта в порядке возрастания их заработной платы, полученной в январе, (первая строка должна соответствовать тому участнику проекта, который получил наименьшую заработную плату в январе). Для решения этой задачи нужно выполнить сортировку двухмерного массива, в ходе которой необходимо переставлять строки массива таким образом, чтобы, в итоге, числа в первом столбце расположились в возрастающем порядке.

Представим, что нам необходимо расположить помесячную заработную плату Иванова в порядке возрастания ее значений (первый столбец должен соответствовать месяцу, в котором Иванов получил наименьшую заработную плату). Эта задача также решается путем сортировки двухмерного массива. Однако, теперь в ходе сортировки мы должны переставлять столбцы и делать это таким образом, чтобы, в итоге, числа в первой строке располагались в возрастающем порядке.

Сортировку элементов заданной строки или столбца двухмерного массива можно выполнять любым из методов сортировки одномерных массивов. Например, одним из рассмотренных ранее: методом «пузырька», методом прямого выбора или методом вставки.

Разработаем алгоритм сортировки двухмерного массива на примере сортировки рассмотренного выше массива, хранящего сведения о заработной плате пяти участников проекта за шесть месяцев работы. Рассмотрим случай, когда элементы заданной строки располагаются в порядке возрастания их значений, а в качестве метода сортировки используется метод «пузырька». Модель приведена на рис. 8.43.

В начальной части алгоритма выполняются подготовительные действия, необходимые для последующей сортировки. Эти действия моделируются несколькими, последовательно расположенными структурами типа блок. Вначале объявляется, создается и инициализируется двухмерный массив `salary` (заработная плата), хранящий значения заработной платы участников проекта. Массив состоит из пяти строк и шести столбцов. В каждой строке хранятся значения заработной платы одного человека. Столбцы хранят значения заработных плат, полученных всеми участниками проекта с января по июнь включительно. Затем выполняются действия по выводу на экран монитора массива `salary` с исходным расположением элементов. Следующий блок объявляет и инициализирует переменную с именем

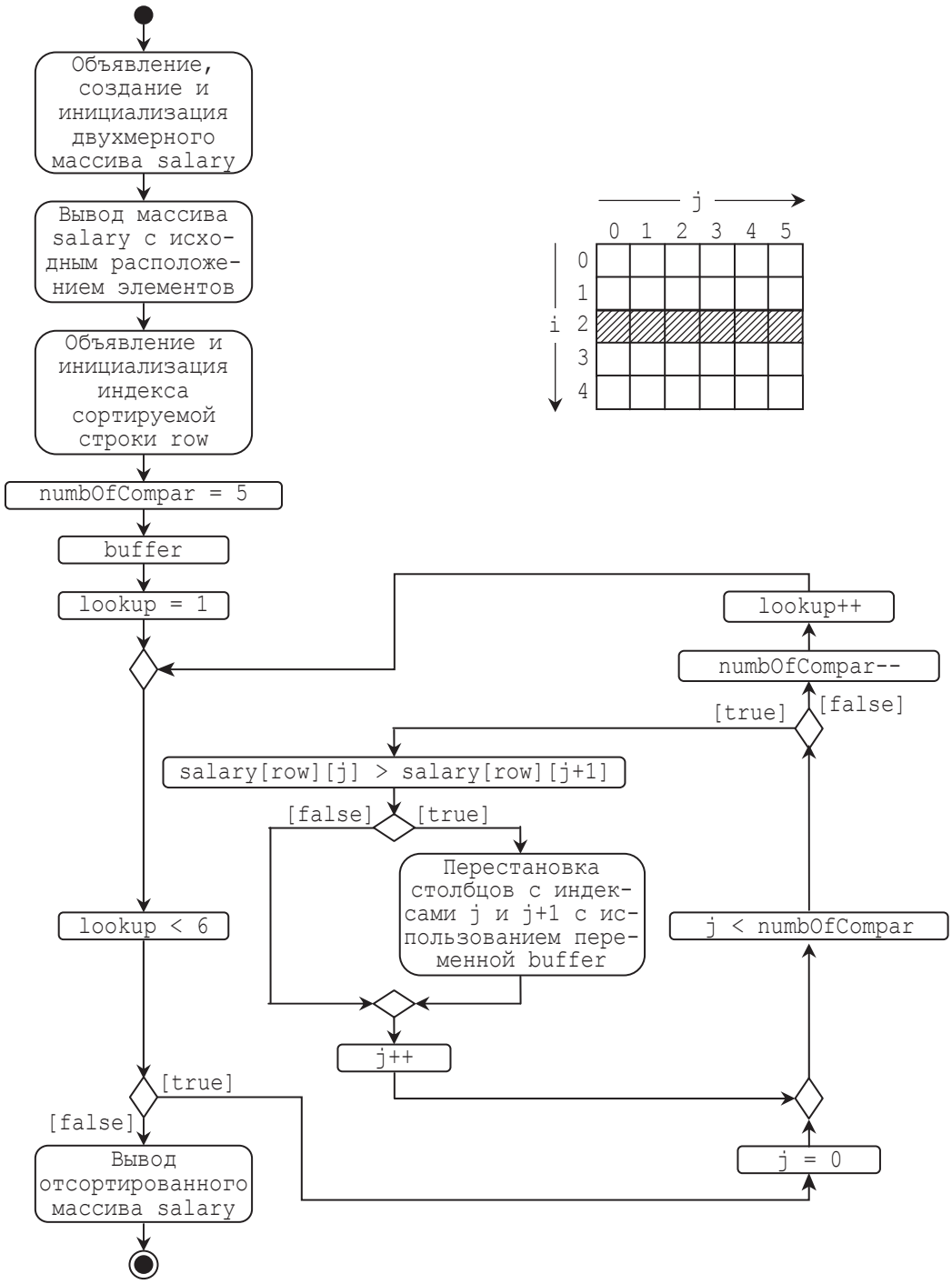



**Рис. 8.43.** Модель алгоритма сортировки заданной строки двухмерного массива методом «пузырька» в порядке возрастания значений. В верхней правой части двухмерный массив `salary` представлен графически. Отмечена сортируемая строка

numbOfCompar (количество сравнений), которая хранит количество сравнений пар смежных чисел в сортируемой строке, выполняемых при каждом просмотре этой строки. Эти сравнения необходимы для реализации метода «пузырька». Затем следует блок, который объявляет переменную с именем row (строка), предназначенную для хранения индекса строки, подлежащей сортировке. Эта переменная должна быть проинициализирована, например, путем ввода с клавиатуры. Завершает цепочку действий, необходимых для последующей сортировки, блок, который объявляет переменную buffer (буфер). Буфер необходим для реализации действий по перестановке элементов смежных строк. Способ перестановки элементов массива с использованием буферной переменной мы рассмотрели ранее (см. рис. 8.17).

Центральная часть алгоритма представляет собой вложенные циклы типа «вначале проверка условия, затем итерация», осуществляющие сортировку строки с индексом row методом «пузырька». Метод пузырька предполагает многократный просмотр сортируемой строки. Общее количество просмотров на единицу меньше, чем количество элементов в строке. В нашем примере строка состоит из шести элементов, и, следовательно, для завершения сортировки понадобится пять просмотров. Параметром внешнего цикла является переменная с именем lookup (просмотр), являющаяся счетчиком просмотров. Эта переменная инициализируется единицей и перед началом очередной итерации увеличивается на единицу. Условие окончания итераций записано в виде выражения lookup < 6, которое обеспечивает выход из внешнего цикла после выполнения пяти итераций.

Внутренний цикл управляет действиями, выполняемыми при каждом просмотре строки с индексом row в соответствии с методом «пузырька». Применительно к нашему случаю эти действия можно описать следующим образом. Начиная с первой пары чисел в строке с индексом row, необходимо попарно сравнить все смежные числа. Если в текущей паре чисел число в предыдущей колонке оказывается больше, чем число в последующей колонке, то выполняется перестановка этих колонок. Каждая итерация внутреннего цикла устанавливает на «правильное место» наибольшее из чисел из не отсортированной части строки. Поэтому количество сравнений в каждой последующей итерации на единицу меньше, чем количество сравнений в предыдущей итерации.

Параметром внутреннего цикла является переменная j, являющаяся одновременно и переменной-индексом колонки массива salary. Параметр внутреннего цикла инициализируется нулем, а условие окончания итераций во внутреннем цикле в виде выражения j < numbOfCompar обеспечивает выход из цикла в том случае, когда значение параметра j станет равным значению переменной numbOfCompar. После каждого выхода из внутреннего цикла значение переменной numbOfCompar уменьшается на единицу.

Телом внутреннего цикла является структура развилка типа «одно условие — один блок». Условие ветвления этой структуры записано в виде булева выражения salary[row][j] > salary[row][j+1], задающего условие сравнения пары смежных чисел в сортируемой строке. Если это выражение принимает значение true (предыдущее число больше последующего), то выполняется структура типа блок, обеспечивающая действия по перестановке столбцов с индексами j и j+1.

В заключительной части алгоритма размещена структура типа блок, действия которой осуществляют вывод на экран монитора отсортированного массива.

На рис. 8.44 приведен фрагмент программного кода, в который может быть отображена модель, приведенная на рис. 8.43.

```
double[][] salary = {{8000.50, 7555.00, 4987.50, 6003.00, 5897.60, 8100.00},
                     {3111.20, 2002.75, 1880.00, 2003.00, 3897.60, 2670.90},
                     {1050.15, 1002.50, 1580.10, 1990.00, 1000.00, 1703.50},
                     {7700.00, 8002.40, 7500.45, 8080.70, 7000.35, 7003.00},
                     {7700.00, 8002.40, 7500.45, 8080.70, 7000.35, 7003.00}};

//вывод массива salary с исходным расположением элементов

int row;

//ввод значения row

int numbOfCompar = 5;
double buffer;

for(int lookup = 1; lookup < 6; lookup++){
   for(int j = 0; j < numbOfCompar; j++)
      if(salary[row][j] > salary[row][j+1]){
         //перестановка столбцов с индексами j и j+1 с использованием buffer
      }
   }
   numbOfCompar--;
}
//вывод отсортированного массива
```

**Рис. 8.44.** Отображение алгоритма сортировки двухмерного массива на рис. 8.43 в программный код

Код, приведенный на рис. 8.44, состоит из предложений и операторов, которые неоднократно использовались в ранее описанных примерах, и должен быть понятен, по крайней мере, тем, кто изучал предыдущий материал. Поэтому отметим только особенности этого кода.

Двухмерный массив `salary` объявляется, создается и инициализируется при помощи одного предложения, которое записано таким образом, чтобы легко воспринимались строки и столбцы матрицы.

Для организации вложенных циклов используется оператор `for`, а не оператор `for-each`, поскольку в алгоритме необходимо манипулировать индексами массива.

В коде комментариями отмечены места, где должны быть записаны предложения, кодирующие вывод двухмерного массива на монитор и перестановку столбцов двухмерного массива с заданными значениями индексов.

Возможны различные варианты вывода массива salary на экран монитора. Простейший вариант предполагает вывод только значений элементов массива в виде матрицы, в таком же виде, как она представлена в коде на рис. 8.44. В более сложном варианте, кроме матрицы значений элементов массива, выводятся наименования строк и столбцов. Строки могут именоваться фамилиями и инициалами участников проекта, а столбцы — наименованиями месяцев. Действия, осуществляющие вывод элементов двухмерного массива, выполняются вложенными циклами с традиционным распределением функций между внешним и внутренним циклами. Внешний цикл управляет переходом от предыдущей строки к последующей, а внутренний — выводит на экран элементы текущей строки. Однако, поскольку на экран выводятся все элементы массива и, следовательно, требуется систематический просмотр всех элементов, то для организации как внешнего, так и внутреннего циклов можно использовать оператор for-each. На рис. 8. 45 приведен фрагмент кода, осуществляющий вывод массива salary на экран монитора.

Итерационной переменной внешнего цикла является одномерный массив с именем row, в который на каждой итерации внешнего цикла копируется очередная строка массива salary. Итерационной переменной внутреннего цикла является переменная с именем x, в которую на каждой итерации внутреннего цикла копируется очередной элемент массива row. Телом внутреннего цикла является предложение System.out.print(x + ", "), которое последовательно выводит в одну строку на экране монитора значения текущей строки массива salary.

```
for(double row[] : salary){
   for(double x : row)
      System.out.print(x + ", ");
   System.out.println();
}
```

**Рис. 8.45.** Фрагмент кода, осуществляющего вывод двухмерного массива salary на экран монитора

После выхода из внутреннего цикла и перед началом очередной итерации внешнего цикла осуществляется перевод курсора на начало следующей строки при помощи предложения System.out.println().

Перестановку элементов столбцов с индексами j и j+1 в строке i с использованием переменной buffer можно осуществить при помощи итерационного процесса путем многократного выполнения следующих предложений.

```
buffer = salary[i][j];
salary[i][j] = salary[i][j+1];
salary[i][j+1] = buffer;
```

Количество повторных выполнений приведенных предложений равно количеству строк массива, и, следовательно, в ходе итерационного процесса

переменная-индекс строки `i` должна изменяться в пределах от 0 до 4. На рис. 8.46 приведен фрагмент кода, осуществляющего перестановку элементов столбцов двухмерного массива с индексами `j` и `j+1`.

```
for(int i = 0; i < salary.length; i++){
   buffer = salary[i][j];
   salary[i][j] = salary[i][j+1];
   salary[i][j+1] = buffer;
 }
```

**Рис. 8.46.** Фрагмент кода, осуществляющего перестановку элементов столбцов двухмерного массива с индексами `j` и `j+1`

## Упражнения для самостоятельной работы

8.1. Разработайте программный код, который для одномерного массива положительных целых чисел, состоящего из *n* элементов, вычисляет сумму произведений первого элемента на последний, второго на предпоследний и т. д.

8.2. Разработайте программный код, который для массива действительных чисел вычисляет значение многочлена степени n в точке $x_0$, считая элементы массива коэффициентами этого многочлена.

8.3. Разработайте программный код, который для одномерного массива действительных чисел, состоящего из *n* элементов, вычисляет сумму всех элементов массива так, чтобы четные элементы входили в нее с обратными знаками.

8.4. Разработайте программный код, который для массива действительных чисел, состоящего из *n* элементов, находит количество элементов, превосходящих среднеарифметическое значение элементов массива.

8.5. Разработайте программный код, который для массива положительных целых чисел `a[]`, состоящего из *n* элементов, проверит, выполняется ли для элементов массива следующее отношение: `a[i] <= a[i+1] <= a[i+2]`, и выведет на экран консоли соответствующее сообщение.

8.6. Разработайте программный код, который для массива действительных чисел, состоящего из *n* элементов, выполняет следующие действия: (1) находит минимальный и максимальный элемент; (2) сортирует элементы массива, расположенные между минимальным и максимальным элементами, по убыванию их значений методом пузырька.

8.7. Разработайте программный код, который для массива целых чисел, состоящего из *n* элементов, выполняет следующие действия: (1) находит номер первого и последнего четного элемента; (2) сортирует элементы массива, расположенные между первым и последним четными элементами, по убыванию их значений методом прямого выбора.

8.8. Разработайте программный код, который перестроит числовой массив, состоящий из *n* элементов, так, чтобы вначале подряд (в том же порядке) следовали отрицательные значения его элементов, затем нули, а затем положительные значения.

8.9. Разработайте программный код, который для целой квадратной матрицы *n*-го порядка определяет, является ли она магическим квадратом, т. е. такой, в которой суммы элементов во всех строках и столбцах одинаковы.

8.10. Разработайте программный код, который для целой квадратной матрицы *n*-го порядка упорядочивает значения элементов главной диагонали по алгоритму вставки, а значения побочной диагонали — по алгоритму пузырька.

8.11. Разработайте программный код, который для целой квадратной матрицы *n*-го порядка осуществляет обход матрицы по спирали по часовой стрелке, начиная с ее левого верхнего угла, и выводит на экран элементы матрицы в порядке их обхода.

8.12. Разработайте программный код, который преобразует числовую матрицу так, чтобы строки с нечетными индексами были упорядочены по убыванию, а с четными — по возрастанию.

8.13. Разработайте программный код, который для числовой квадратной матрицы *n*-го порядка проверяет ее симметричность относительно главной диагонали. На консоль должно быть выведено слово "YES", если матрица симметрична, либо слово "NO" в противном случае.

8.14. Разработайте программный код, который для числовой квадратной матрицы *n*-го порядка поменяет местами элементы, стоящие на главной и побочной диагонали, при этом каждый элемент должен остаться в том же столбце (то есть в каждом столбце нужно поменять местами элемент на главной диагонали и на побочной диагонали).

8.15. Разработайте программный код, который сортирует строки числового двухмерного массива так, чтобы первой шла строка, сумма элементов которой была меньше, чем остальных, и так далее, по возрастанию. Используйте алгоритм сортировки выбором.

ЧАСТЬ 2

# ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ МОДЕЛИРОВАНИЕ

Раздел 9

# ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ ПАРАДИГМА ПРОГРАММИРОВАНИЯ

Во второй части книги рассматриваются вопросы моделирования программных систем средствами *унифицированного языка моделирования* UML (Unified Modeling Language) и *языка объектных ограничений* OCL (Object Constraint Language). Значительная часть унифицированного языка моделирования UML и языка объектных ограничений OCL могут рассматриваться как формализованное изложение объектно-ориентированной парадигмы программирования (ООП). Сложно понять UML-модели и OCL-предложения без знакомства с основными идеями, лежащими в основе этой парадигмы. Поэтому девятый раздел представляет собой введение в базовые понятия и идеи, составляющие ООП. В настоящем разделе крайне редко используется UML и вовсе не используется OCL. Иногда используется простой Java-подобный псевдокод, который позволяет без излишней детализации записывать компактные предложения, необходимые для иллюстрации излагаемых идей.

## 9.1. Концептуальные основы

Начнём изучение ООП с перечисления основных понятий, составляющих её концептуальную основу. Вначале только назовём эти понятия, сформировав необходимый словарь, поскольку их углублённому изучению посвящено последующее изложение.

Одним из фундаментальных понятий ООП является понятие *программного объекта*. Объект — это элементарный «действующий компонент» системы. Будем считать, что любая сущность может мыслиться как объект, а любая программная система может быть представлена набором взаимодействующих объектов. Техническое устройство или его отдельный компонент, персонаж компьютерной игры, отдельная личность, банковский счёт и т. п. могут быть представлены в виде объектов. Объект обладает внутренней структурой и состоит из двух частей: (1) набора *полей* и (2) набора *методов*.

Набор полей будем называть *атрибутивной моделью* объекта. Отдельное поле объекта, в общем случае, также является объектом. В этом смысле атрибутивная модель некоторого объекта является объединением (ассоциацией) объектов. Количество объектов-полей, ассоциированных в атрибутивную модель, и их смысловая нагрузка определяются контекстом функционирования объекта. Например, если моделируется система, предназначенная для учёта подержанных автомобилей, предназначенных для продажи, то объектами такой системы могут быть продаваемые автомобили. Атрибутивные модели этих объектов могут быть составлены из следующих полей: «тип автомобиля», «характеристики двигателя», «год выпуска», «количество пройденных километров», «данные о предыдущем владельце», «стоимость» и т. д. Любой объект может быть представлен различным количеством атрибутов. Чем конкретнее объект (чем больше мы знаем об объекте), тем большим количеством атрибутов мы можем его описать. Например, мы можем легко продолжить список атрибутов объекта «автомобиль» для приведенного выше примера. Однако, чем более абстрактным и неопределенным является объект (чем меньше мы знаем об объекте), тем меньшим количеством атрибутов мы можем его описать. Например, сложно составить длинный список атрибутов для модели такого объекта, как «душа».

Поля, входящие в атрибутивную модель, могут быть сгруппированы по общности характеристик. В дальнейшем мы рассмотрим классификацию полей по отношению к различным классификационным признакам, а пока введём классификацию, необходимую для дальнейшего изложения. Будем делить поля на: (1) *базовые* и (2) *производные*. Значения базовых полей не зависят друг от друга, а значения производных полей формируются из значений базовых полей. В приведенном выше перечне полей атрибутивной модели автомобиля все поля являются базовыми, поскольку, например, значение поля «тип автомобиля» не зависит от значения поля «стоимость». Но если мы включим в атрибутивную модель автомобиля поле «возраст автомобиля», то это поле будет производным, поскольку его значение формируется из значения поля «год выпуска». Другой пример. Если среди полей атрибутивной модели объекта, моделирующего конкретную личность, имеются базовые поля: «имя», «отчество» и «фамилия», то поле «инициалы» является производным, поскольку его значение формируется из значений перечисленных базовых полей. Главное отличие производных полей от базовых заключается в том, что значения производных полей не могут быть изменены произвольным образом, а должны изменяться синхронно с изменением соответствующих базовых полей. Поэтому производные поля имеют статус полей, предназначенных только для чтения.

Совокупность значений полей объекта называется *состоянием объекта*. Значения полей объекта в процессе его функционирования может изменяться. Поэтому состояние объекта определяется моментом времени.

Вторым структурным компонентом объекта является набор методов. Методы определяют функциональные возможности объекта или, как часто говорят, *поведение объекта*. Метод представляет собой программный код, оформленный в виде функции. *Неявным параметром методов объекта является сам объект*, поскольку

выполнение какого-либо из методов объекта, как правило, изменяет состояние этого объекта. ООП предполагает, что состояние объекта может быть изменено только при помощи его методов.

Метод объекта начинает выполняться после получения запроса от другого объекта. Запрос, инициирующий выполнение метода, называется *сообщением*. Таким образом, в основе функционирования объектной системы лежит обмен сообщениями между её объектами. Сообщение всегда адресовано конкретному методу, принадлежащему конкретному объекту. Для того, чтобы сообщение могло быть адресовано конкретному объекту, он должен иметь уникальный *объектный идентификатор*.

Каждый объект уникален, поскольку каждый объект уникально характеризуется своим состоянием и идентификатором, и в то же время, каждый объект принадлежит *классу объектов*. Таким образом, ООП предполагает, что в мире, в котором мы живём, все сущности представлены не единичными экземплярами, а множествами экземпляров или классами. Поэтому иногда объект класса называется также, *экземпляром класса*. Класс имеет примерно такую же структуру, как и объект, т. е. состоит из: (1) полей и (2) методов. Однако, если объект — это «действующий элемент» системы, который характеризуется значениями своих полей (состоянием) и может выполнять действия при помощи своих методов, то класс — это описание множества структурно подобных объектов. Понятие состояние не применимо к классу. Класс не может выполнять никаких действий. Программист описывает класс объектов, а не каждый объект в отдельности. ООП предполагает, что после того как класс описан, то с его помощью можно создать сколь угодно много структурно подобных объектов, задавая полям, описанным в классе, конкретные значения.

При моделировании программных систем в такой формализованной среде, как UML-OCL, «размывается» граница между этапом анализа предметной области и этапом проектирования, и, по сути, оба эти этапа реализуются в процессе моделирования. Так, например, банковская система по обслуживанию клиентов включает кассиров, клиентов, банковские счета, банковские операции, денежные средства и т. д. Все эти сущности реальной системы в объектной программе будут представлены соответствующими программными объектами, объединёнными в классы. В модели рассматриваемого примера, наверное, будут фигурировать классы: Кассир, Клиент, Банковский счёт, Банковская операция, Денежные средства и т. д.

Возможность создания классов, моделирующих конкретную проблемную область, является одним из достоинств ООП. Объявление классов вводит в объектную программу *новые типы*, расширяющие язык программирования в направлении моделируемой системы или проблемы. Идея в том, чтобы позволить разработчику программы адаптировать язык программирования к моделируемой системе или проблеме путём введения новых типов, специфичных для этой системы или проблемы. ООП ориентирует программиста на описание проблемы в терминах самой проблемы, а не в терминах примитивных данных, отражающих архитектуру компьютера.

Часто объекты классифицируют как *постоянные* и *непостоянные*. Местом хранения непостоянных объектов является основная, энергозависимая память компьютера, а местом хранения постоянных объектов — внешняя, энергонезависимая память. Программная система, которая строится из непостоянных объектов, называется *объектная программа*, а программная система, которая строится из постоянных объектов — *объектная база данных*. Модели, синтезированные в среде UML-OCL, как правило, могут быть трансформированы либо в объектную программу, либо в объектную базу данных.

ООП представляет собой совокупность идей о том, как должны строиться программная система, язык программирования и система программирования, позволяющие создавать хорошо структурированные, легко модифицируемые и надёжные компьютерные программы. Среди множества идей, лежащих в основе ООП, выделим и рассмотрим следующие:

- инкапсуляция;
- память объекта о своих предыдущих состояниях;
- объектная идентичность;
- сообщения;
- статические поля и методы;
- наследование и
- полиморфизм.

### 9.1.1. Инкапсуляция

Объектная *инкапсуляция* — это такой способ внутренней организации объекта, когда состояние объекта может быть доступно или модифицировано только опосредованно, при помощи собственных методов этого объекта. Идея инкапсуляции базируется на предположении о том, что непосредственный доступ к атрибутивной модели должен быть запрещён. В этом случае объект контролирует доступ к своим полям, и получить этот доступ можно только при помощи методов объекта. Инкапсуляция защищает атрибутивную модель объекта и детали реализации его методов от несанкционированного вмешательства извне и упрощает работу с объектом.

Для того, чтобы поручить объекту выполнить какую-то работу, нет необходимости быть знакомым с его атрибутивной моделью и деталями реализации методов. Для этого достаточно знать назначение методов и уметь их вызывать. Инженеры, конструирующие технические системы, сплошь и рядом используют идею инкапсуляции. Так, например, мы легко «поручаем» телевизору уменьшить громкость или переключиться на другой канал, не зная ни внутреннего устройства телевизора, ни того, как он это делает.

Для наглядной графической иллюстрации идеи инкапсуляции изобразим объект в виде атрибутивной модели, окружённой «защитным слоем» методов, так, как это изображено на примере объекта `dwarf` (гном) на рис. 9.1.

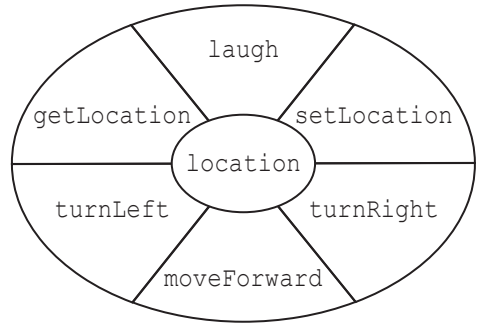



**Рис. 9.1.** Инкапсулированный объект `dwarf`

Объект, изображенный на рис. 9.1, моделирует фигурку гнома, перемещающуюся по экрану монитора. Монитор рассматривается как сетка, состоящая из множества клеток, регламентирующих перемещение объекта-гнома. Объект может поворачиваться направо или налево, перемещаться вперёд на некоторое количество клеток и смеяться.

Опишем атрибутивную модель и методы объекта `dwarf` в нотации, принятой в UML.

Атрибутивная модель объекта `dwarf` описывается только одним полем.

```
location:Square
```

Поле `location` хранит информацию о текущем местонахождении объекта. Тип с именем `Square` вводится программистом, например, путём объявления одноимённого класса.

Поведение объекта `dwarf` определяется набором следующих методов.

```
turnRight():void
```

Метод `turnRight` поворачивает объект на 90° градусов по ходу часовой стрелки. Метод не имеет входных параметров и ничего не возвращает.

```
turnLeft():void
```

Метод `turnLeft` поворачивает объект на 90° градусов против хода часовой стрелки. Метод также не имеет входных параметров и возвращаемого значения.

```
moveForward(numbOfSquares:int):boolean
```

Метод `moveForward` перемещает объект вперёд на некоторое количество клеток. Метод имеет один входной параметр с именем `numbOfSquares`, который задаёт это количество клеток. Метод возвращает значение типа `boolean`, которое информирует об успешном или неуспешном выполнении метода.

```
laugh():void
```

Метод `laugh` издаёт звук, похожий на смех гнома. Он не имеет входных параметров и возвращаемого значения.

```
getLocation():Square
```

Метод `getLocation` позволяет узнать местоположение объекта на экране монитора. Он не имеет входных параметров, но возвращает значение типа `Square`.

```
setLocation(newLocation:Square):void
```

Метод `setLocation` позволяет установить фигурку гнома в любую клетку, задаваемую параметром `newLocation` (новое местонахождение).

В дальнейшем мы подробно рассмотрим, каким образом в UML описываются поля и методы и как их можно уточнить при помощи OCL-ограничений. Пока договоримся о следующем. При описании имен примитивных типов данных в UML моделях будем использовать типы, принятые в языке программирования Java. В простейшем случае описатель поля состоит из имени поля и имени типа поля, разделённые двоеточием, а описатель метода — только из его заголовока в виде имени метода, перечня входных параметров в круглых скобках, двоеточия и типа возвращаемого значения. Если у метода отсутствуют входные параметры, то круглые скобки остаются пустыми. Если метод ничего не возвращает, то в конце, после двоеточия, записывается служебное слово `void` (пустой). Имена полей класса, входных параметров метода и локальных переменных метода, а также имена методов начинаются со строчной (маленькой) буквы. Имена типов переменных и возвращаемых значений будем записывать следующим образом. Если используется примитивный тип, то его имя начинается со строчной буквы (например, `int`, `boolean`) Если программист вводит новый тип, то имя этого типа начинается с прописной буквы (например, `Square`). В том случае, если имя переменной или метода состоит из нескольких слов, например `getLocation` (получить местонахождение), то слова, из которых оно состоит, записываются без пробела и каждое новое слово, кроме первого, начинается с прописной (большой) буквы.

Инкапсулированный объект скрывает свою атрибутивную модель от пользователя объекта. Полное или частичное сокрытие атрибутивной модели называется *информационной скрытностью*. Термин информационная скрытность означает, что пользователь объекта не только не имеет доступа к полям объекта, но также не знает их количество, имена и типы. Это, однако, не мешает пользователю использовать объект, заставляя его выполнять работу при помощи своих методов, в том случае, если программисту известно, какими методами обладает объект и как ими пользоваться. Если имеет место информационная скрытность, то изменение состояния объекта невидимо для пользователя. Так, например, не зная атрибутивную модель объекта, приведенного на рис. 9.1, можно заставить этот объект повернуть налево, направо или переместиться вперёд. Несмотря на информационную скрытность, пользователь может получить доступ к атрибутивной модели объекта, если в список методов объекта введены специальные методы доступа. Например, значение поля

location можно прочесть при помощи метода getLocation (получить местонахождение), а при помощи метода setLocation (установить местонахождение) можно записать в это поле новое значение.

Инкапсуляция предполагает также *реализационную скрытность*. Реализационная скрытность означает, что от пользователя объекта скрывается то, какую внутреннюю организацию имеют методы объекта и как они реализованы. Например, метод moveForward (передвинуть вперёд) позволяет переместить гнома вперёд на некоторое количество клеток, задаваемое значением аргумента numbOfSquares (количество клеток) в виде целого положительного числа. Однако это не обязательно означает, что в методе moveForward местонахождение объекта представлено таким же способом. Это может быть, например, пара переменных (xCoor, yCoor) для случая прямоугольной системы координат или какой-либо другой способ описания местоположения объекта. Внутреннее представление местонахождения гнома в методе moveForward скрыто от пользователя объекта.

Информационная и реализационная скрытности являются способами «сворачивания» сложности программной системы. По сути, они означают, что для внешнего пользователя объект может быть представлен в виде «чёрного ящика». Внешний пользователь обладает знаниями о том, что может делать объект, но не знает, как он это делает и как он внутренне сконструирован. Таким образом, для внешнего пользователя полностью инкапсулированный объект представляется так, как это схематически изображено на рис. 9.2.

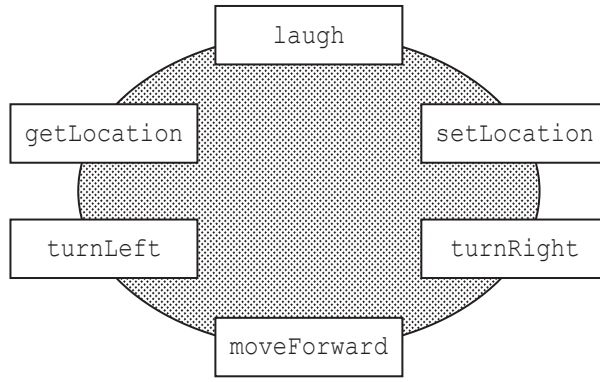



**Рис. 9.2.** Инкапсулированный объект dwarf, как он представляется внешнему пользователю

Представление объекта в виде «чёрного ящика» означает, что внешнему пользователю объекта доступен только его *интерфейс в виде набора заголовков методов*.

Инкапсуляция обеспечивает программе следующие преимущества. Во-первых, ограничивается влияние локальных проектных решений только на данный объект. Имеется в виду, что изменения, вносимые в объект, не оказывают влияния на остальную часть программы.

Во-вторых, разделяются содержание информации и форма её представления. Пользователю объекта не доступна форма представления информации внутри объекта. Это защищает объект от несанкционированного внешнего вмешательства в его внутреннюю организацию.

### 9.1.2. Память объекта о своих предыдущих состояниях

Объект обладает способностью запоминать своё предыдущее состояние. Когда обычная программная процедура или функция завершает работу и возвращает управление вызвавшей его программе, она «умирает», оставляя после себя только полученный результат. Когда эта же функция вызывается повторно, то она заново «рождается», не помня ничего из своей предыдущей «жизни». Например, каждый раз, когда вызывается функция нахождения значения синуса некоторого аргумента, в переменную, в которой накапливается окончательный результат, должен быть записан ноль, а в переменную, которая определяет момент выхода из цикла — фиксированное значение, определяющее точность вычисления синуса.

Когда создаётся и размещается в памяти вновь созданный объект, то в его поля записываются некоторые начальные значения, определяющие начальное состояние объекта. Этот процесс называется *начальной инициализацией объекта*. В дальнейшем, в процессе функционирования объекта, его состояние изменяется, однако объект, в отличие от программной функции, помнит своё прошлое и сохраняет информацию о своём предыдущем состоянии неопределённо долго. Совокупность полей объекта (его атрибутивная модель) представляет собой то «запоминающее устройство», в котором хранится предыдущее состояние объекта. В приведенном выше примере объекта, моделирующего фигурку гнома, перемещающегося по экрану монитора, состояние, описываемое значением поля `location` (местонахождение), после завершения работы какого-либо из методов этого объекта сохраняется внутри объекта. Представим себе, что мы, при начальной инициализации объект `dwarf`, установили гнома в центре экрана, а затем переместили его вперёд, вызвав метод `moveForward`. Гном занял новое положение. Представим далее, что после этого мы прервали работу с гномом, вызвали метод другого объекта, но через некоторое время вернулись к гному и снова вызвали метод `moveForward`. После повторного вызова метода `moveForward` гном не «прыгнет» в центр и не начнёт движение из центра, а переместится в новое положение, начиная от того положения, которое он занимал после первого вызова метода `moveForward`.

### 9.1.3. Объектная идентичность

Объектная идентичность — это свойство объекта, которое позволяет идентифицировать его как отдельную и уникальную программную сущность и отличать от всех остальных объектов. Для реализации объектной идентичности

каждый объект должен содержать что-то уникальное, что отличает его от всех остальных объектов.

Существует два способа реализации объектной идентичности. Первый способ основан на том, что объекты различаются по их состояниям. Объекты одного и того же класса отличаются друг от друга тем, что значения их полей различно. В противном случае мы имели бы идентичные копии одного и того же объекта. Таким образом, совокупное значение всех полей однозначно идентифицирует объект. В ряде случаев для однозначной идентификации объекта достаточно знать значения только некоторых его полей. В частном случае для этого достаточно знать значение только одного поля. Например, объекты класса `Person` (личность) могут однозначно идентифицироваться тремя полями: `firstName` (имя), `secondName` (фамилия) и `dateOfBirth` (дата рождения). Однако, можно ввести в список полей класса `Person` поле, значение которого достаточно для уникальной идентификации личности. Это может быть, например, поле, хранящее индивидуальный идентификационный номер государственной налоговой инспекции. Совокупность полей класса, значения которых достаточны для уникальной идентификации объекта этого класса, называется *ключом класса*. Ключ доступен программисту, и он может прочитать его значение. Таким образом, первый способ реализации объектной идентичности заключается в задании ключа класса при его объявлении и в использовании этого ключа при работе с объектами. Такой способ реализации объектной идентичности применяется, главным образом, при моделировании баз данных.

Второй способ реализации объектной идентичности, который используется, главным образом, в объектных программах, заключается в том, что уникальность объекта обеспечивается специальным полем, называемым объектным идентификатором (сокращённо OID — Object Identifier), которое *автоматически вводится в объект в момент его создания*. OID является внутренним средством идентификации объектов, он недоступен программисту и, с точки зрения компилятора, является адресом участка основной памяти компьютера, в котором размещается объект. В момент создания объекта его OID записывается в переменную ссылочного типа. Для работы с объектом программисту не нужно знать значение OID. Достаточно знать имя ссылочной переменной. Созданием объектов управляет программист, включая в код метода специальные предложения со служебным словом `new`. Ранее мы использовали предложения со служебным словом `new` для создания объектов, при изучении классов-оболочек (см. подраздел 3.4 ) и массивов (см. подраздел 8.1). Объект класса `Dwarf` может быть создан при помощи следующего предложения:

```
Dwarf dwr1 = new Dwarf(dwr1Location);
```

Предложение означает, что необходимо создать объект класса `Dwarf` и записать его OID в переменную с именем `dwr1` типа `Dwarf`. Вновь созданный объект необходимо разместить в клетку, задаваемую значением `dwr1Location`. Механизм автоматической идентификации объектов при помощи OID гарантирует, что: (1) объект

сохраняет один и тот же идентификатор во время своего существования, вне зависимости от того, что происходит с объектом; (2) не существует двух объектов, которые имеют один и тот же идентификатор; (3) всякий раз, когда создаётся новый объект, ему приписывается идентификатор, который отличается от всех остальных идентификаторов, как созданных в прошлом, так и тех, которые будут созданы в будущем. На рис. 9.3 изображён объект, снабженный объектным идентификатором в виде условного значения 123456.

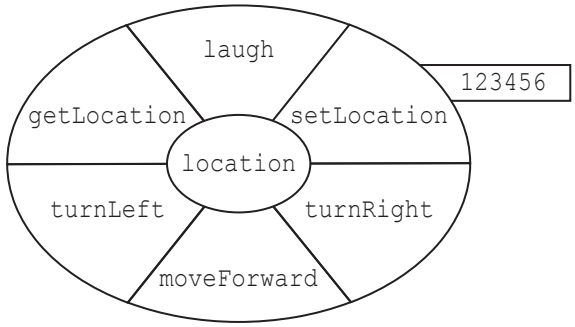



**Рис. 9.3.** Объект класса `Dwarf` с OID, равным 123456

Как было отмечено ранее, для получения доступа к объекту не надо знать значение OID. Программист получает доступ к объекту опосредованно, через значение ссылочной переменной. Рис. 9.4 иллюстрирует отмеченный опосредованный доступ.

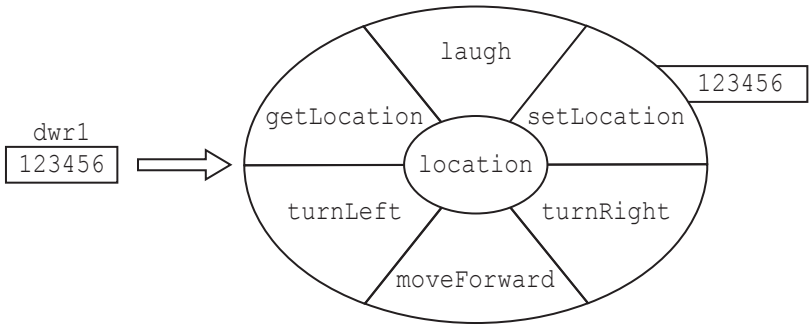



**Рис. 9.4.** Значение переменной `dwr1` ссылается на объект с OID, равным 123456

Ясно, что выполнение предложения

```
Dwarf dwr2 = new Dwarf(dwr2Location)
```

порождает ещё один объект, также принадлежащий классу `Dwarf`, но с другим идентификатором, например, 654321. Рис. 9.5 иллюстрирует доступ к объекту `dwr2`.

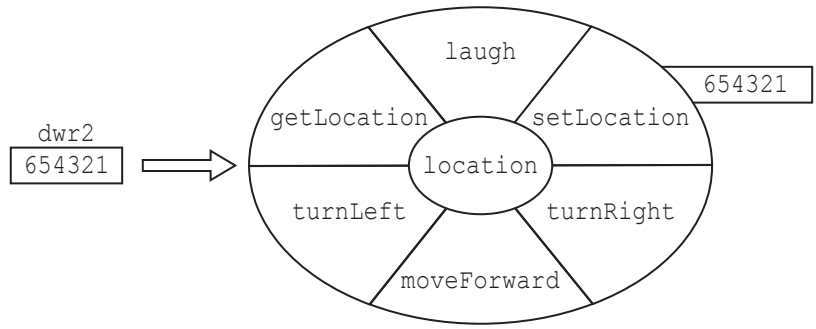



**Рис. 9.5.** Значение переменной `dwr2` ссылается на объект с OID, равным 654321

Если мы теперь выполним операцию присваивания:

```
dwr2 = dwr1
```

то значения обеих переменных, `dwr1` и `dwr2`, будут одинаковы, обе переменные будут указывать на один и тот же объект, имеющий идентификатор, равный 123456 (значение переменной `dwr1`), и, следовательно, доступ к объекту с идентификатором 654321 (значение переменной `dwr2`) становится невозможным. Эта ситуация иллюстрируется рис. 9.6.

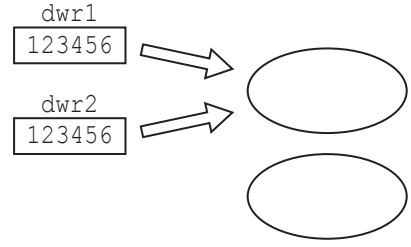



**Рис. 9.6.** Переменные `dwr1` и `dwr2` ссылаются на один и тот же объект. Доступа ко второму объекту более не существует

В языке программирования Java объекты, на которые отсутствуют ссылки, рассматриваются как ненужные и подлежащие уничтожению. Такие объекты, как правило, автоматически уничтожаются при помощи программы, называемой «сборщик мусора». Программа «сборщик мусора» периодически сканирует память, обнаруживает объекты, на которые отсутствуют ссылки, и освобождает память от них. Следовательно, если программа написана на языке программирования Java, то для удаления ненужных объектов не надо принимать специальных мер. Достаточно сделать так, чтобы на ненужные объекты отсутствовали ссылки.

Для различения одноименных полей в различных объектах, созданных при помощи одного и того же класса, необходимо использовать *составные имена полей*. Составное имя поля объекта состоит из имени ссылочной переменной на объект и

имени поля, разделенные символом «точка». Таким образом, структура составного имени поля объекта имеет вид:

```
<имя ссылки на объект> . <имя поля>
```

Вернемся к ситуации, когда созданы и размещены в памяти два различных объекта `dwr1` и `dwr2` класса `Dwarf` и нашей целью является кодирование метода, в котором необходимо сравнить местоположение гномов. Местоположение гномов хранится в двух полях с одинаковым именем `location`. Однако, для корректной записи выражения, осуществляющего сравнение, нам необходимы различные имена для переменных, хранящих местоположение первого и второго гномов. Поэтому в упомянутом выражении необходимо использовать составные имена:

`dwr1.location` поле `location` в объекте `dwr1`,
`dwr2.location` поле `location` в объекте `dwr2`.

Составные имена необходимы не только для различения одноименных полей в объектах, созданных при помощи одного и того же класса, но во всех случаях, когда код метода использует имена полей «своих» и «чужих» объектов.

### 9.1.4. Сообщения

Объектно-ориентированная программа реализует своё поведение путём последовательной активизации объектов. Активный объект выполняет один из своих методов. Объект запрашивает другой объект о выполнении некоторого метода при помощи *сообщения*. Сообщение может также служить переносчиком данных от одного объекта к другому. ООП допускает, чтобы объект передавал сообщения самому себе. Таким образом, сообщение является средством, при помощи которого *объект-отправитель* передаёт *объекту-получателю* запрос о выполнении одного из методов объекта-получателя. Объектом-отправителем и объектом-получателем может быть один и тот же объект. Объект-отправитель называют также *отправителем* или *клиентом*, а объект-получатель — *мишенью* или *сервером*.

Для того, чтобы объект-отправитель мог сформировать сообщение, адресованное объекту-мишени, он должен «знать» три компонента сообщения.

1. Имя ссылочной переменной на объект-мишень.
2. Имя метода объекта-мишени, который он желает вызвать.
3. Значения параметров вызываемого метода, если таковые имеются.

Первый компонент необходим для того, чтобы выбрать один конкретный объект из множества объектов программы, второй — для того, чтобы выбрать один конкретный метод из множества методов объекта-мишени, а третий — для того, чтобы передать в вызываемый метод фактические значения параметров.

Примером простого сообщения, при помощи которого вызывается метод, не содержащий входных параметров, может служить сообщение

```
dwr1.turnRight()
```

Здесь переменная `dwr1` содержит ссылку на объект-мишень, а `turnRight` — метод объекта-мишени, который необходимо выполнить. Точка используется в качестве разделителя между именем ссылочной переменной и именем метода.

Примером сообщения, при помощи которого вызывается метод, содержащий входные параметры, может служить сообщение

```
dwr1.moveForward(numbOfSteps)
```

Здесь переменная `dwr1`, как и в предыдущем примере, содержит ссылку на объект-мишень, `moveForward` (передвинуть вперёд) — имя метода объекта-мишени, `numbOfSteps` (количество шагов) — значение входного параметра.

Когда в классе объявляется метод с параметрами, то в его заголовке указывается список параметров, называемых формальными. Каждый элемент списка *формальных параметров* представляет собой пару: имя формального параметра и его тип. Когда при помощи сообщения вызывается метод, то в сообщении указываются значения параметров, которые называются *фактическими параметрами*. Фактические параметры могут представлять собой константы, переменные или выражения. Если фактический параметр задан переменной, то её имя может не совпадать с именем соответствующего формального параметра. Важно, чтобы типы формального и фактического параметров были одинаковыми или совместимыми. В заголовке метода `moveForward` указано имя формального параметра `numbOfSquares`, а при его вызове мы, в качестве фактического параметра, использовали переменную с именем `numbOfSteps`.

Ясно, что один и тот же объект в одном случае может быть отправителем, а в другом — мишенью. Поэтому понятия объект-отправитель и объект-мишень являются относительными и определяются по отношению к конкретному сообщению.

Напомним, что понятие объект является обобщением понятия данное. Мы можем рассматривать, например, данные целого типа как объекты класса целых чисел. Поэтому в объектно-ориентированных языках программирования могут отсутствовать данные. В этом случае как поля объекта, так и параметры методов (в том числе и те, которые передаются при помощи сообщений) всегда являются ссылками на объекты. Примером объектно-ориентированного языка программирования, в котором отсутствуют данные, может служить язык Smalltalk. Часто объектно-ориентированные языки программирования являются гибридными в том смысле, что они оперируют как данными (встроенными в язык программирования), так и объектами. Примером гибридного языка программирования является

Java. В полностью объектно-ориентированном языке программирования, таком как Smalltalk, в сообщении

```
dwr1.moveForward(numbOfSteps)
```

переменная `dwr1`, интерпретируется как ссылка на объект-цель, а переменная `numbOfSteps` — как ссылка на объект-параметр метода `moveForward`.

Для дальнейшего изложения нам понадобится классификация сообщений, которая разделяет все сообщения на три группы по отношению к тому, какие методы вызываются с их помощью. Эти группы сообщений носят наименования:

- *инструктивные;*
- *запросные* и
- *императивные.*

*Инструктивное сообщение — это сообщение, при помощи которого вызывается метод, осуществляющий обновление значения поля объекта*.

Значения полей объекта могут изменяться «естественным» образом, в процессе его функционирования. При помощи инструктивного сообщения значение соответствующего поля *обновляется безусловно*. Необходимость в безусловном обновлении значения поля может возникнуть в том случае, если его предыдущее значение «устарело» и требует замены. Например, поле `secondName` (фамилия) класса `Person` (личность) должно быть изменено, если конкретная личность изменила фамилию. Примером инструктивного сообщения может быть сообщение

```
book1.setPrice(newPrice)
```

Сообщение адресовано методу `setPrice` (установить цену) объекта `book1`, передаёт этому объекту инструкцию о том, что цена книги должна быть обновлена и определяет новую цену при помощи значения параметра `newPrice` (новая цена).

Инструктивные сообщения вызывают специальные методы, которые называются стандартные set-методы. Эти методы предназначены для безусловного обновления значений полей. Подчеркнём, что стандартные set-методы осуществляют безусловное обновление полей, поскольку значения полей могут изменяться также естественным образом, в процессе «жизнедеятельности» объекта. Например, в рассмотренном ранее примере с фигуркой гнома, перемещающейся по клеткам, выполнение метода `moveForward` (передвинуть вперёд) естественным образом изменяет значение поля `location` (местонахождение). Но если это необходимо, то мы можем принудительно изменить значение этого поля и заставить гнома «прыгнуть» на некоторую клетку при помощи set-метода `setLocation` (установить местонахождение). Не все поля могут быть изменены безусловно и, следовательно, не ко всем полям применимы стандартные set-методы. Стандартные set-методы не могут применяться к тем полям, которые должны оставаться неизменными на протяжении всей «жизни» объекта. Например, после создания объекта класса `Person` (личность) не может изменяться значение поля `dateOfBirth` (дата рождения) этого объекта.

Стандартные set-методы не могут также применяться к производным полям, поскольку значения производных полей не могут изменяться безусловно, а зависят от значений базовых полей.

*Запросное сообщение — это сообщение, при помощи которого вызывается метод, осуществляющий чтение значения поля объекта*. Примером запросного сообщения может быть сообщение

```
dwr1.getLocation()
```

Это сообщение запрашивает объект `dwr1` о его текущем местонахождении. Характерной чертой запросного сообщения является то, что оно ничего не меняет в состоянии объекта. При помощи запросных сообщений вызываются специальные методы, называемые стандартными get-методами. Эти методы, в отличие от стандартных set-методов, применимы для всех полей. При помощи запросных сообщений могут вызываться также get-методы, которые часто называются запросными. Отличие запросного get-метода от стандартного get-метода заключается в том, что он не предназначен для чтения значения поля, а возвращает значение, сформулированное в запросе. Например, класс `Room` (комната) может содержать поля, хранящие данные о размерах комнаты, а также о расположении окон и дверей. К объекту такого класса можно сформировать запросное сообщение, вызывающее метод-запрос `getWallsArea` (найти площадь стен), который возвращает суммарную площадь стен с учетом оконных и дверных проемов. При этом в классе `Room` может отсутствовать производное поле `wallsArea` (площадь стен) для хранения значения, возвращаемого методом `getWallsArea`.

*Императивное сообщение — это сообщение, при помощи которого вызывается метод, реализующий одно из возможных поведений объекта*. Примером императивного сообщения может быть рассмотренное ранее сообщение

```
dwr1.moveForward(numbOfSteps)
```

являющееся командой на перемещение объекта `dwr1` вперёд на некоторое количество клеток, задаваемых значением параметра `numbOfSteps` (количество шагов).

Модели управляющих программ оперируют большим количеством императивных сообщений. Например, модель программы управления промышленным роботом может оперировать следующим сообщением

```
robotManipulator.positioning(x, y, z, alpha1, alpha2, alpha3)
```

Приведенное сообщение требует, чтобы манипулятор робота (объект `robotManipulator`), при помощи метода `positioning` (позиционирование) был установлен в точку пространства, в соответствии со значениями параметров, задающих значения шести степеней свободы манипулятора.

### 9.1.5. Размещение класса в памяти. Статические поля и методы

*Класс представляет собой модель, которая используется для создания объектов*. Каждый новый объект, создаваемый при помощи некоторого класса, имеет один и тот же набор полей и методов, определённых при объявлении этого класса. Таким образом, объекты одного и того же класса структурно идентичны, но, как выяснилось выше, имеют следующие отличия:

- каждый объект имеет свой, уникальный идентификатор (OID);
- каждый объект находится в уникальном состоянии, которое определяется значениями его полей для данного момента времени.

Таким образом, *класс — это программная сущность, которую программист проектирует и описывает в своей программе, а объект — это программная сущность, которая создается и «работает» в процессе выполнения объектной программы*. Поэтому термин объектно-ориентированное программирование не совсем точен. Более точное наименование, соответствующее ООП, — это *класс-структурированное программирование*.

Проводя аналогию между производством изделий методом штамповки и созданием объектов в объектной программе, мы можем сказать, что в объектной программе класс играет роль штампа, при помощи которого «производятся» объекты. Создание нового объекта осуществляется при выполнении специального предложения, включающего служебное слово `new` и рассмотренное ранее.

Рассмотрим структуру класса с точки зрения вариантов его размещения в основной памяти компьютера. Это позволит нам ввести важную классификацию полей и методов класса.

На рис. 9.7 схематически представлено простейшее размещение в памяти двух объектов, созданных при помощи одного и того же класса.

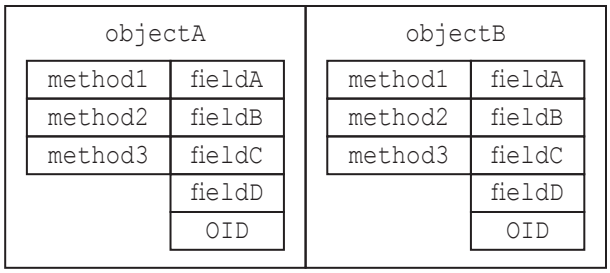



**Рис. 9.7.** Простейшее размещение в памяти двух объектов одного класса. OID — объектный идентификатор

Видно, что способ размещения объектов в памяти, иллюстрируемый рис. 9.7, нерационально расходует её пространство. Каждый объект хранит один и тот же набор методов. Например, код метода `method1` абсолютно идентичен в обоих объектах.

Понятно, что каждый объект должен хранить свой индивидуальный набор полей и индивидуальный объектный идентификатор, поскольку они определяют объектную идентичность и уникальное состояние объектов.

Однако, если методы работают сугубо последовательно и в каждый момент времени может выполняться код только одного метода, то можно предложить более рациональное размещение класса в основной памяти. Такое рациональное размещение предполагает, что в памяти хранится только один набор методов, который используется всеми объектами класса *в режиме разделения времени процессора*. На рис. 9.8 схематически представлено более рациональное размещение в памяти этих же объектов.

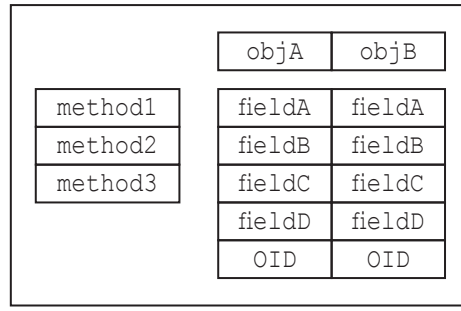



**Рис. 9.8.** Рациональное размещение двух объектов в памяти

Поля и методы, о которых мы говорили до сих пор, моделируют атрибуты и поведение объектов класса. Однако, каждый класс является самостоятельной сущностью, которая также обладает своими собственными атрибутами и поведением, не зависящими от атрибутов и поведения его объектов. Для описания атрибутов и поведения самого класса используются специальные поля и методы, называемые *статическими*. У каждого класса имеется один неизменный набор статических полей и методов вне зависимости от того, сколько объектов создано при помощи этого класса.

Примером статического поля может служить поле `numbOfObjects` (количество объектов), хранящее общее количество объектов, созданных при помощи данного класса. Значение этого поля должно увеличиваться на единицу каждый раз, когда создаётся новый объект, и уменьшаться на единицу каждый раз, когда из памяти удаляется один из объектов. Ясно, что поле `numbOfObjects` характеризует весь класс, а не его отдельный объект. Со статическими полями могут работать только статические методы. В общем случае статические методы реализуют те элементы поведения класса, которые присущи классу как отдельной сущности. Рис. 9.9 построен на базе рис. 9.8 и иллюстрирует размещение в памяти двух объектов, а также статические поля и методы этого класса.

Отметим ещё одно примечательное свойство статических членов класса. Количество объектов некоторого класса в процессе работы объектной программы

в общем случае не остаётся неизменным. Новые объекты создаются и размещаются в памяти, а ненужные — удаляются. Поэтому общее количество нестатических полей и методов изменяется. Количество же статических полей и методов всегда неизменно.

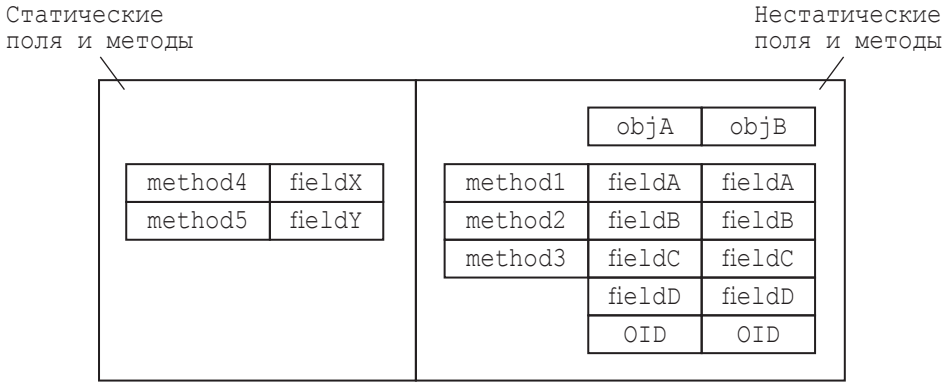



**Рис. 9.9.** Статические и нестатические поля и методы

В дальнейшем в процессе изучения предмета и углубления наших знаний о полях и методах мы будем рассматривать всё новые и новые их виды. Сейчас, подводя предварительный итог, отметим, что поля бывают: (1) базовыми или производными, (2) статическими или нестатическими. Что касается методов, то мы классифицировали их как: (1) стандартные (set- и get-методы) или нестандартные, а также (2) статические или нестатические.

### 9.1.6. Наследование

В практике разработки объектно-ориентированных программ встречается задача, заключающаяся в том, что необходимо разработать новый класс, который незначительно (всего на несколько дополнительных полей и/или методов) отличается от ранее описанного класса. Простейшее решение заключается в том, что при описании нового класса в него копируются все поля и методы старого класса, а затем добавляются дополнительные. Однако, более рациональным решением является такой способ описания класса, который позволяет ему автоматически использовать поля и/или методы ранее описанного класса.

Наследованием (из класса `Donor` в класс `Acceptor`) называется способ описания системы классов, при помощи которого в классе `Acceptor` (получатель) неявно определяются поля и методы класса `Donor` (донор) таким образом, как если бы они были определены непосредственно в классе `Acceptor`. В этом случае класс `Donor` называют *суперклассом*, а класс `Acceptor` — *подклассом*. Другими словами, если между классами `Donor` и `Acceptor` установлено такое отношение, что

класс Donor является суперклассом, а класс Acceptor — его подклассом, то объекты класса Acceptor могут использовать поля и методы, объявленные в классе Donor. Наследование «работает» только в одном направлении. *Подкласс может наследовать поля и методы у суперкласса, но суперкласс не может наследовать поля и методы у подкласса*.

Наследование является характерной чертой ООП, которая отличает его от других парадигм программирования. Наследование придаёт ООП несколько привлекательных свойств. Например, идея наследования позволяет легко разрабатывать новую версию программы на базе существующей версии. Новые версии представляют собой, как правило, расширенные варианты старых версий. Расширение класс-структурированной программы осуществляется путём создания подклассов, расширяющих возможности классов предыдущей версии на основе наследования.

Идея наследования предполагает, что программная система или её часть представляет собой иерархию классов. В верхней части иерархии располагаются классы, реализующие общие атрибуты и поведение системы, а в нижней части — классы, реализующие её специфические атрибуты и поведение.

Рассмотрим пример. Пусть в некотором приложении, связанном с морскими перевозками, имеется класс морских судов Vessel. Этот класс может быть определён при помощи поля course (курс) и метода turn (повернуть на заданный курс). Поле course имеет тип Angle (угол), а метод turn обладает одним входным параметром newCourse (новый курс) типа Angle и возвращает значение типа boolean. Будем считать, что таким способом класс Vessel реализует наиболее общие атрибуты и поведение морских судов. Однако, существуют специализированные морские суда, требующие для своего описания дополнительные поля и методы. Например, специфической особенностью подводной лодки является её способность погружаться. Таким образом, мы можем определить ещё один класс Submarine (подводная лодка), который наследует члены класса Vessel, дополняя их своими специфическими членами, например, полем depth (глубина) и методом submerge (погрузиться на заданную глубину). Рис. 9.10 иллюстрирует графическое представление иерархической соподчинённости классов Vessel и Submarine.

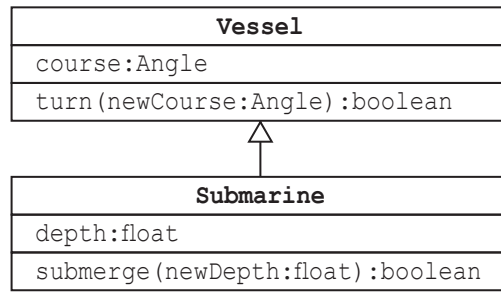



**Рис. 9.10.** Класс Submarine является подклассом класса Vessel и наследует его поля и методы

Рис. 9.10 является примером UML-модели. Диаграмма, изображенная на рис. 9.10, называется диаграммой классов и моделирует структуру простой программной системы, состоящей из классов `Vessel` и `Submarine`. На диаграмме классов класс изображается при помощи графического символа класса, который представляет собой прямоугольник, разделенный на три отделения горизонтальными линиями. В верхнем отделении записывается имя класса, во втором отделении специфицируется атрибутивная модель, а в третьем — список методов класса. Графический символ, представляющий собой стрелку с полым треугольником на конце, является графическим символом, при помощи которого на рис. 9.10 моделируется наследование из класса `Vessel` в класс `Submarine`.

Рассмотрим, как работает механизм наследования, путём анализа приведенного ниже фрагмента псевдокода, в котором вначале создаются объекты классов `Vessel` и `Submarine`, а затем им последовательно передаются четыре императивных сообщения.

```
Vessel vs = new Vessel();
Submarine sb = new Submarine();

vs.turn(newCourse);          (1)
sb.submerge(newDepth);       (2)
sb.turn(newCourse);          (3)
vs.submerge(newDepth);       (4)
```

Проанализируем сообщения (1)–(4) с точки зрения их выполнимости.

(1) Объект `vs` получает сообщение, при помощи которого ему передаётся требование выполнить метод `turn` (повернуть на заданный курс). Поскольку объект `vs` создан при помощи класса `Vessel`, то объект `vs` просто использует метод `turn`, определённый в этом классе, и сообщение (1) будет выполнено.

(2) Объект `sb` получает сообщение, при помощи которого ему передаётся требование выполнить метод `submerge` (погрузиться на заданную глубину). Поскольку объект `sb` создан при помощи класса `Submarine`, то он использует метод `submerge`, определённый в классе `Submarine`. Сообщение (2) будет также выполнено.

(3) Объект `sb` получает сообщение, при помощи которого ему передаётся требование выполнить метод `turn`. Без наследования это сообщение являлось бы причиной прерывания работы программы, поскольку `sb` является объектом класса `Submarine`, в котором метод `turn` не определен. Однако поскольку `Vessel` является суперклассом для `Submarine`, то объект `sb` имеет право на использование любого метода класса `Vessel`. Поэтому сообщение (3) успешно выполнится.

(4) Это сообщение не будет выполнено. Объект `vs` создан при помощи класса `Vessel`, в котором отсутствует метод `submerge`. Наследование в этом случае неправомерно, поскольку действует в направлении от суперкласса к

подклассу (в направлении, противоположном стрелке на рис. 9.10), но не наоборот. Таким образом, попытка выполнения сообщения (4) вызовет прерывание работы программы.

Объект класса иногда называют *экземпляром* класса и часто понятия «объект» и «экземпляр» используют как синонимы. Однако это не всегда правомерно. Наследование позволяет дифференцировать понятия объект и экземпляр, поскольку допускает, что *некоторый объект может быть одновременно экземпляром более одного класса*. В этом смысле понятие объекта является более общим и включает в себя понятие экземпляра класса. В нашем примере объект, созданный при помощи класса `Submarine`, является не только экземпляром этого класса, но также экземпляром класса `Vessel`, поскольку наследует его поля и методы.

Существует так называемый «*является* тест» (*is a test*), при помощи которого можно проверить корректность приписывания классам статуса «суперкласс» и «подкласс». Если предложение: «некоторый `A` *является* `D`», семантически корректно, тогда `A` является подклассом `D`. Например, поскольку предложение «подводная лодка *является* морским кораблём» семантически корректно, то класс `Submarine` является подклассом класса `Vessel`. Ясно, что предложение «морской корабль *является* подводной лодкой» семантически некорректно и класс `Submarine` не является подклассом класса `Vessel`.

На рис. 9.11 показано, каким образом в памяти компьютера размещаются поля и методы объекта класса `Submarine` (подводная лодка).

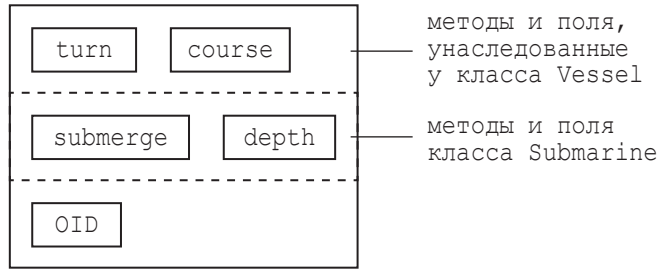



**Рис. 9.11.** Размещение в памяти объекта класса `Submarine`

Объект некоторого класса может наследовать поля и методы у объектов нескольких классов. При *множественном наследовании* каждый класс может иметь произвольное количество суперклассов. На рис. 9.12 приведена диаграмма классов, иллюстрирующая пример множественного наследования.

В диаграмме классов на рис. 9.12 используется простой графический символ класса, состоящий из одного отделения с именем класса.

Реализация идеи множественного наследования классов в объектно-ориентированных языках программирования связана с необходимостью решения ряда проблем, среди которых проблема конфликта имён полей и методов суперклассов. Если, например, в объектах суперклассов имеются поля или методы с

одинаковыми именами, то объект подкласса должен уметь определять, из какого именно суперкласса необходимо наследовать поле или метод. Не все объектно-ориентированные языки программирования реализуют множественное наследование классов. Например, множественное наследование классов не реализовано в языке программирования Java.

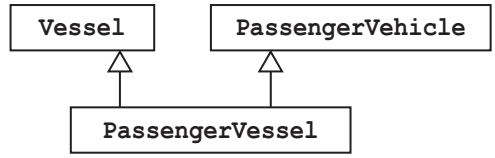



**Рис. 9.12.** Множественное наследование. Подкласс `PassengerVessel` (пассажирское судно) наследует поля и методы у двух суперклассов `Vessel` и `PassengerVehicle` (пассажирское транспортное средство)

### 9.1.7. Полиморфизм

Слово «полиморфизм» имеет греческое происхождение и составлено из двух слов, означающих соответственно «много» и «форма». Быть полиморфным означает обладать свойством, принимать различные формы. Полиморфизм можно также понимать как свойство некоторой сущности объектной системы в различных ситуациях вести себя так, как будто она принадлежит к нескольким различным типам. ООП предполагает, что полиморфными могут быть различные сущности объектно-ориентированной системы. В настоящем параграфе мы ограничимся изучением полиморфных методов и определим полиморфизм метода при помощи двух взаимосвязанных утверждений.

(A) Полиморфизм метода означает, что в нескольких классах могут быть определены методы, имеющие одинаковый заголовок, но различный код.

(B) Полиморфизм метода предполагает, что одна ссылочная переменная в различные моменты времени ссылается на объекты различных классов.

Поясним, как работают эти утверждения на примере. Предположим, что имеется класс `PlaneFigure` (плоская фигура), который представляет собой множество плоских фигур: прямоугольников, окружностей, треугольников и т. д. Для класса `PlaneFigure` естественными являются поле `perimeter` (периметр) и метод `getPerimeter` (получить периметр), который возвращает значение периметра для объекта класса `PlaneFigure`. Ясно, что код метода `getPerimeter` должен быть достаточно сложным, поскольку он должен уметь вычислять периметр любой плоской фигуры.

Введём в систему несколько подклассов класса `PlaneFigure`: класс `Triangle` (треугольник), класс `Rectangle` (прямоугольник) и класс `Circle` (окружность). Это действительно подклассы класса `PlaneFigure`, поскольку и треугольник, и прямоугольник, и окружность *являются* плоскими фигурами (вспомним «*is a test*»).

На рис. 9.13 приведена диаграмма классов образовавшейся системы.

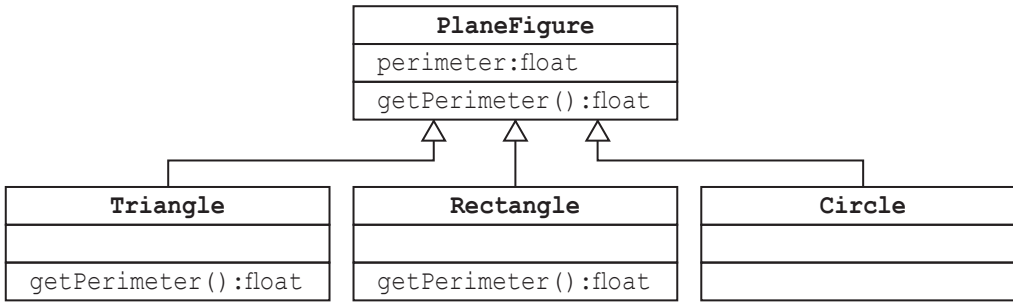



**Рис. 9.13.** Класс `PlaneFigure` и его подклассы

Как видно на рис. 9.13, некоторые из вновь введенных подклассов также содержат метод `getPerimeter`, который выполняет ту же работу, что и метод `getPerimeter` в классе `PlaneFigure` — вычисляет периметр плоской фигуры, но специализирован на вычислении периметра одного класса плоских фигур. Программный код метода `getPerimeter` для класса `Rectangle`, например, сильно отличается от программного кода метода `getPerimeter` для класса `PlaneFigure`. Вычисление периметра прямоугольника выполняется очень просто, поэтому код метода `getPerimeter` класса `Rectangle` также прост. Поскольку алгоритм вычисления периметра любой произвольной плоской фигуры, определённой в суперклассе, сложен, то целесообразно ввести в подклассы несколько полиморфных методов `getPerimeter` с простыми алгоритмами.

Приведенный на рис. 9.13 пример иллюстрирует определение (A), утверждающее, что полиморфизм метода означает, что в нескольких классах определён метод, имеющий одинаковый заголовок, но функционирующий различным образом в каждом из классов. Проиллюстрируем теперь определение (B). Если в систему введено несколько полиморфных методов, то необходимо уметь программировать логику вызова этих методов. Например, необходимо, чтобы код мог определить момент, когда необходимо вычислить периметр конкретной плоской фигуры, например, прямоугольника, и в этот момент вызывал метод `getPerimeter`, определённый в классе `Rectangle`. Определение (B) подсказывает каким образом можно вызвать необходимый полиморфный метод при помощи единственного сообщения. Мы можем составить следующее сообщение

```
figure.getPerimeter()
```

при помощи которого в разные моменты времени будем вызывать один из требуемых полиморфных методов `getPerimeter`. Для этого нужно сделать так, чтобы ссылочная переменная `figure` в разные моменты времени ссылалась на разные объекты.

Возможны пять различных случаев вызова метода `getPerimeter`, в зависимости от значения переменной `figure`.

1. Переменная figure содержит ссылку на объект класса Triangle. В этом случае будет вызван метод getPerimeter, определённый в классе Triangle.
2. Переменная figure содержит ссылку на объект класса Rectangle. В этом случае будет вызван метод getPerimeter, определённый в классе Rectangle.
3. Переменная figure содержит ссылку на объект класса Circle. Метод getPerimeter в этом классе не определён, но поскольку класс Circle является подклассом класса PlaneFigure, метод getPerimeter будет унаследован из класса PlaneFigure и выполнен.
4. Переменная figure содержит ссылку на объект класса PlaneFigure. В этом случае будет выполнен метод getPerimeter, определённый в классе PlaneFigure.
5. Переменная figure содержит ссылку на объект класса Vessel, которого нет в системе, изображенной на рис. 9.13, следовательно, вызов метода не будет выполнен и компилятор сообщит о синтаксической ошибке.

Сообщение figure.getPerimeter() означает, что объект-отправитель посылает сообщение, «не зная», в каком объекте-мишени, будет вызван метод. Использование такого сообщения является часто применяемым приёмом при работе с полиморфными методами.

Рассмотрим следующий фрагмент псевдокода.

```
PlaneFigure figure;
Triangle t = new Triangle();
Circle c = new Circle();
. . .
if(<пользователь выбрал треугольник>)
   figure = t;
else
   figure = c;
. . .
figure.getPerimeter();          // figure ссылается на объекты
                                // классов Triangle или Circle
. . .
```

В приведенном псевдокоде нет необходимости осуществлять проверки с целью определения того, какая версия метода getPerimeter должна быть выполнена. Вместо этого необходимо правильно выбрать объект-мишень. Говоря метафорически, объект-мишень «знает», как вычислить требуемый периметр, и поэтому объект-отправитель не должен «беспокоиться» об этом. Приведенное в начале фрагмента кода предложение

```
PlaneFigure figure;
```

ограничивает полиморфизм в рассматриваемом примере, в том смысле, что переменной figure разрешено ссылаться только на объекты класса PlaneFigure или на

объекты его подклассов. Если переменной `figure` будет когда-либо присвоена ссылка, например, на объект класса `Person`, то программа остановится.

Метод `getPerimeter`, определённый в нескольких классах, соответствует понятию полиморфного метода в смысле определения (A). Переменная `figure`, указывающая на объекты различных классов, соответствует понятию полиморфизма в смысле определения (B). Весь пример показывает, что оба определения полиморфного метода работают совместно.

## Упражнения для самостоятельной работы

9.1. Расширьте атрибутивную модель объекта `dwarf` (см. рис. 9.1). Представьте её списком, состоящим из пяти полей. Приведите примеры нескольких состояний для этого объекта.

9.2. Определите понятие «инкапсуляция». Нужно ли вводить в список методов объекта специальные методы, обеспечивающие инкапсуляцию? Приведите пример, иллюстрирующий ваш ответ.

9.3. Представьте книгу в виде класса с именем `Book` и со следующими полями: `title` (заголовок); `author` (автор); `publisher` (издатель); `year` (год издания); `numOfPages` (количество страниц); `price` (цена). Предложите набор методов для этого класса. Введите в список полей одно статическое поле.

9.4. Представьте личность в виде трёх классов, предназначенных для использования в различных системах. Класс `Person1` используется в системе кадрового учёта предприятия, класс `Person2` — в системе учёта успеваемости студентов в деканате университета, класс `Person3` — в системе учёта клиентов онлайнового магазина. Опишите каждый из этих классов при помощи наборов полей.

9.5. Какие способы реализации объектной идентичности вы знаете? Приведите примеры трёх ключей для класса `Person1`, полученного при выполнении упражнения 9.4. Запишите несколько примеров предложений для создания объектов класса `Person1`.

9.6. Объясните, каким образом цель моделирования оказывает влияние на описание полей и методов классов. Приведите примеры, подтверждающие ваши выводы.

9.7. Проиллюстрируйте свойство наследования на примере системы, состоящей из следующих классов: `Doctor` (врач); `Therapeutist` (терапевт);

`Surgeon` (хирург). Нарисуйте диаграмму, связывающую эти классы, используя нотацию рисунка 9.13. Отметьте, какие члены (поля и методы) суперкласса могут наследоваться.

9.8. Введите в подклассы системы, предложенной в 9.7, полиморфные методы. Объясните смысл двух определений полиморфизма на примере рассматриваемой системы.

9.9. Предложите набор полей и методов для класса, моделирующего автоматические стиральные машины и их основные функции. Предложите примеры трёх типов сообщений (инструктивное, запросное и императивное), адресованные объекту этого класса.

9.10. Рассмотрите систему, состоящую из классов `Car` (автомобиль) и `Driver` (водитель). Для этой системы: (1) разработайте классы, включающие минимальный набор полей и методов; (2) создайте по одному объекту для каждого классса и (3) запишите примеры трёх типов сообщений (инструктивное, запросное и императивное) для случая, когда отправителем является объект класса `Car`, и случая, когда отправителем является объект класса `Driver`.

9.11. Университет может быть представлен в виде набора классов. Состав этого набора классов и члены каждого из классов зависят от цели моделирования. Представьте университет в виде наборов классов для следующих случаев: (1) цель моделирования — структура научно-исследовательской деятельности; (2) цель моделирования — структура учебного процесса; (3) цель моделирования — структура зданий и помещений.

9.12. Представьте географические карты в виде класса `Map` со следующими полями: `region` (район); `scale` (масштаб); `listOfCities` (список городов). Предложите набор методов для этого класса. Учтите, что значение поля `listOfCities` зависит от значения поля `scale`. На карте с крупным масштабом указывается меньше городов, чем на карте с мелким масштабом.

9.13. Используя нотацию рисунка 9.13, изобразите диаграмму, моделирующую структурные отношения между классами `Room` (комната); `Kitchen` (кухня); `LivingRoom` (гостиная) и `Bedroom` (спальня). Опишите классы при помощи полей и методов.

9.14. Рисунок 9.12 моделирует систему с множественным наследованием. Дополните каждый из классов, изображённых на этом рисунке, полями и методами. Изобразите размещение в основной памяти объекта класса `PassengerVessel` (пассажирское судно). Проиллюстрируйте проблему конфликта имён для членов суперклассов.

Раздел 10

# ОГРАНИЧЕНИЯ

*Ограничение* — это предложение, которое позволяет уточнить описание элемента объектной системы и определить его сферу применимости. Необходимость введения ограничений при разработке объектно-ориентированных моделей продиктована желанием, с одной стороны, сделать эти модели более адекватными моделируемой системе, а с другой — более определенными и пригодными для автоматической генерации кода. Можно утверждать, что каждая сущность, с которой сталкивается человек в окружающем его мире, имеет ограниченную сферу применимости, или ограничена. Например, температура кипения воды ограничена сверху атмосферным давлением; диапазон представления целого числа в компьютере ограничен количеством байтов, выделяемых для хранения целого числа; возраст бракосочетания для мужчин и женщин ограничен снизу законодательством государства и т. д. Некоторые ограничения настолько важны, что имеют статус законов. Эти законы могут быть законами природы, которые человек способен обнаруживать и формулировать, но не в состоянии отменять, либо законами, установленными людьми, ограничивающими поведение индивидуума в сообществе людей и формулирующими правила общежития.

Ограничения не только делают модель более адекватной моделируемой системе, более точной и детерминированной, но и могут использоваться программистом для контроля правильности работы программной системы. Если в процессе работы программы нарушается некоторое ограничение, то это, как правило, квалифицируется как возникновение *исключительной ситуации*, т. е. ситуации, препятствующей безошибочному функционированию программы. Момент возникновения исключительной ситуации фиксируется, а тип исключительной ситуации используется для обработки исключительной ситуации.

Ограничения могут быть сформулированы с разной степенью неопределённости. Процесс уточнения ограничений часто является итерационным и соответствует процессу уточнения самой модели. На первых этапах разработки модели ограничения формулируются с высокой степенью неопределённости. По мере развития модели и уточнения её элементов, ограничения формулируются более точно и более определённо. Например, на ранних стадиях разработки модели системы электронной коммерции для характеристики покупателя, формирующего заказ, может быть сформулировано ограничение в виде следующего предложения на естественном

языке: «Покупатель не должен иметь существенную задолженность». По мере развития модели это ограничение уточняется и может приобрести следующий вид: «Покупатель не должен иметь: (1) задолженность по кредиту, превышающую оговоренный минимум, (2) просроченные неоплаченные счета».

Ограничения часто формулируются в виде слов или предложений естественного языка, как, например, в приведенном выше примере. Недостатком такого способа формулировки ограничений является высокая степень неоднозначности интерпретации ограничений, сложность отображения ограничения в программный код, а также сложность организации контроля за правильностью работы программы.

Существует искусственный, символьный язык, при помощи которого можно единообразно и строго записывать ограничения, накладываемые на элементы модели. Этот язык носит наименование *объектный язык ограничений* (*Object Constraint Language* или *OCL*).

## 10.1. Использование естественного языка для записи ограничений

Базовые структурные элементы класса, такие как поле и метод, часто требуют уточнения в виде ограничений, которые могут быть представлены фразами или отдельными словами естественного языка. Хороший стиль моделирования предполагает использование английского языка. Ограничение записывается в фигурных скобках и размещается в конце строки, описывающей поле или метод. Таким образом, с учётом ограничения, структура строки, описывающей поле, имеет следующий вид:

```
<имя поля>:<тип поля> {<ограничение>}
```

Например:

```
depth:float {less than or equal to 500 metre}
```

В приведенном примере мы ввели ограничение для поля `depth` (глубина), которое ранее (см. рис. 9.10) использовалось при моделировании класса подводных лодок `Submarine`. Ограничение представляет собой предложение, сформулированное на естественном языке и ограничивающее сверху диапазон значений поля `depth`. Ограничение утверждает, что глубина погружения подводной лодки не может быть равной или превышать 500 метров. Эта информация важна для разработчика программы и позволяет ему организовать контроль значения поля `depth` и, следовательно, контроль глубины погружения подводной лодки. Когда мы говорим о записи ограничений на естественном языке, то это не означает, что мы не имеем права сокращать предложение естественного языка путём использования математических

и логических символов. Например, приведенное выше ограничение поля `depth` может быть записано следующим образом:

```
depth:float {depth < 500}
```

Ниже приведен ещё один пример описания поля, снабженного ограничением:

```
location:Square {initial location in center}
```

Ограничение утверждает, что начальное значение поля `location` (местонахождение) должно быть таким, чтобы фигурка гнома располагалась в центре экрана монитора (см. подраздел 9.1.1). Эта информация важна для разработчика программы и должна использоваться им при разработке кода инициализирующего значения полей объекта.

Аналогичным образом записываются ограничения методов. Поэтому, с учётом ограничения, структура строки, описывающей метод, имеет вид:

```
<имя>(<входн. парам.>):<тип возвр. знач. или void>{<огранич.>}
```

Например:

```
turnRight():void {turn to the right on 90°}
```

Ограничение утверждает, что каждый раз, когда вызывается метод `turnRight` (повернуть направо), он должен осуществлять поворот фигурки гнома направо на 90 градусов (см. подраздел 9.1.1).

В приведенных выше примерах ограничение размещалось непосредственно после того элемента модели, который уточнялся с его помощью. Однако это не единственный способ включения ограничений в модель. Удобно включать ограничения в UML-модель при помощи графического символа комментария, приведенного на рис. 10.1.

**Рис. 10.1.** Графический символ комментария в UML

Комментарий изображается в виде прямоугольника с «загнутым» *верхним правым* углом. Внутри графического символа комментария могут записываться фразы естественного языка или предложения искусственного языка (например, предложения языка объектных ограничений OCL). Фразы естественного языка могут включать математические формулы и выражения.

Символ комментария соединяется с элементом модели, который он комментирует, при помощи пунктирной линии. Рис. 10.2 иллюстрирует использование графического символа комментария для включения ограничений в UML-модель.

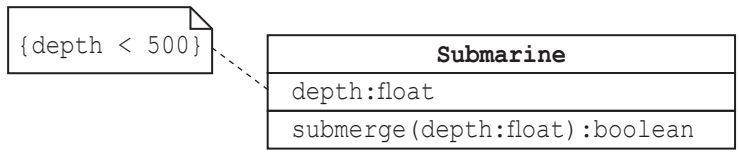



**Рис. 10.2.** Использование графического символа комментария для включения ограничений в UML-модель

Слово или фраза на естественном языке в фигурных скобках, размещённая в верхнем отделении графического символа класса, сразу же за его именем, ограничивает весь класс. На рис. 10.3 приведен пример ограничения всего класса, а не его отдельных членов.

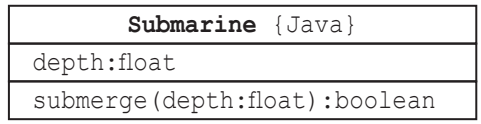



**Рис. 10.3.** Использование естественного языка для ограничения класса

Ограничение `{Java}`, размещенное в верхнем отделении графического символа класс, означает, что при кодировании класса `Submarine` (подводная лодка) должен использоваться язык программирования Java.

## 10.2. OCL-ограничения класса

Использование естественного языка для описания ограничений в модели в ряде случаев достаточно для реализации целей моделирования. Однако, такой способ описания ограничений страдает высокой степенью неопределённости своих формулировок.

Искусственный символьный язык OCL разработан специально для точной и определённой формулировки ограничений и *используется в качестве дополнения к диаграмматическому языку* UML. OCL не предназначен для самостоятельного использования как независимое средство специфицирования объектно-ориентированных систем. В OCL-ограничениях используются имена классов, полей, методов, параметров методов, полюсов ассоциаций и т. д., указанные в UML-модели, которая уточняется при помощи этих ограничений.

Только наиболее простые и примитивные модели могут быть представлены исключительно в виде UML-диаграмм. Более точные модели представляют собой комбинацию UML-диаграмм и OCL-ограничений. OCL является декларативным, а не процедурным языком. Это означает, что при помощи OCL-ограничений описываются «внешние», желаемые характеристики элементов модели, но не описывается то, каким образом эти характеристики могут или должны быть реализованы в коде.

В настоящем разделе рассматриваются те виды OCL-ограничений, которые используются *для уточнения объектов класса, а также полей и методов класса*. Таким образом, целью настоящего раздела является изучение тех средств языка объектных ограничений, которые могут использоваться для уточнения модели, состоящей из одного-единственного класса.

Все виды OCL-ограничений, которые мы будем использовать в настоящем разделе, имеют следующую структуру:

```
context <описание элемента модели, на который распространяется ограничение>
<служебное слово, определяющее вид ограничения>:
        <спецификация ограничения, соответствующая его виду>
```

OCL-ограничение класса, в общем случае, состоит из двух предложений. Первое предложение начинается со служебного слова `context` и задаёт контекст ограничения. При помощи `context`-предложения описывается тот элемент модели, который специфицируется при помощи данного ограничения (класс, поле или метод).

Второе предложение начинается со служебного слова, которое определяет вид ограничения. Например, служебное слово `inv` определяет ограничение, называемое инвариантом, которое ограничивает весь класс, а служебное слово `def` определяет ограничение, при помощи которого в модель вводится новое поле или метод. После служебного слова следует спецификация самого ограничения. Эта спецификация может быть очень простой и состоять, например, из константы, задающей значение некоторого поля, или достаточно сложной, представляющей собой несколько OCL-предложений.

В том случае, когда ограничения включают сложные выражения, их целесообразно прокомментировать на естественном языке. *Строчный комментарий* в OCL представляет собой одну или несколько строк символов (обычно это текст на естественном языке), которые начинаются с символов «- -». Например,

```
- - при формировании возвращаемого значения использованы
- - поля классов Lecturer и Subject.
```

Специфицирование ограничения, как правило, осуществляются при помощи OCL-выражения. Для иллюстрации различных видов OCL-ограничений в настоящем разделе приведены примеры ограничений с OCL-выражениями. Примеры

подобраны таким образом, чтобы понимание OCL-выражений не вызывало трудностей у читателя. С этой целью, при записи OCL-выражений использованы только поля с единичным значением и такие операции OCL, которые аналогичны операциям над данными, принятыми в языке программирования Java. В дальнейшем, по мере необходимости, будут рассмотрены все типы данных, принятые в OCL, и операции над ними.

### 10.2.1. Ограничение классов

При уточнении объектов класса используются следующие OCL-ограничения:

- ограничения, специфицирующие инварианты класса;
- ограничения, определяющие новые производные поля;
- ограничения, определяющие новые методы-запросы к атрибутивной модели.

*Инвариант класса* — это свойство, характеризующее объект класса, оно должно оставаться неизменным в течение времени существования объекта. Например, для класса, который моделирует президентов некоторого государства, инвариантом может быть возраст, который для любого объекта этого класса должен быть равным или превышать тридцать пять лет. Неизменность инвариантов в процессе функционирования системы гарантирует её правильную работу.

*Инварианты записываются в виде булевых OCL-выражений. Инвариант должен быть всегда истинным (за исключением того промежутка времени, когда выполняется конструктор или другой метод класса).* Если инвариант принял ложное значение, то это означает, что в программе возникла исключительная ситуация, препятствующая её нормальной работе.

Ограничения, специфицирующие инварианты класса, имеют следующую структуру:

```
context <имя класса>
inv <необязательное имя инварианта>:
    <булево OCL-выражение, определяющее первый инвариант>
inv <необязательное имя инварианта>:
    <булево OCL-выражение, определяющее второй инвариант>
. . .
```

При помощи инвариантов класса, как правило, ограничивается его атрибутивная модель, поэтому булево выражение, определяющее инвариант класса, часто включает имена полей класса. Однако это не абсолютное правило. Булево выражение инварианта может включать также значения, возвращаемые методами, которые неявно задаются их сигнатурами.

Рассмотрим несколько примеров ограничений, специфицирующих инварианты класса. Ограничение значения поля `depth`, класса `Submarine`, пример которого приведен на рис. 9.10, можно использовать в качестве инварианта для всего класса

Submarine. Ниже приведен пример OCL-ограничения, специфицирующего один из возможных инвариантов класса Submarine.

```
context Submarine
inv: depth < criticalDepth
```

Смысл приведенного ограничения в том, что глубина погружения любой подводной лодки из класса Submarine на протяжении всего времени её существования должна быть меньше критической глубины criticalDepth (например, 500 метров). Ограничение состоит из двух предложений. Первое предложение задает контекст в виде имени класса Submarine. Второе предложение задаёт инвариант и поэтому начинается со служебного слова inv, после которого располагается инвариант в виде булевого выражения. Разделителем между служебным словом inv и логическим выражением является двоеточие. В приведенном примере отсутствует имя инварианта.

В OCL имеется служебное слово self, которое используется для того, чтобы указать на принадлежность какого-либо элемента OCL-предложения к контексту. Служебное слово self можно использовать, например, для того, чтобы показать, что инвариант применим к любому из объектов класса, который он ограничивает. Использование служебного слова self особенно актуально в тех случаях, когда в OCL-выражении используются не только члены класса, указанного в контексте, но и члены других классов программной системы. Приведенное выше ограничение может быть записано в следующем виде:

```
context Submarine
inv: self.depth < 500
```

Если класс Submarine включает стандартный метод getDepth, то в булевом выражении приведенного ограничения вместо имени поля depth можно указать сигнатуру этого метода. Например,

```
context Submarine
inv: self.getDepth() < 500
```

Один и тот же класс можно ограничить несколькими инвариантами. Это можно сделать при помощи одного ограничения с несколькими inv-предложениями.

```
context Book
inv: self.firstName = 'Lev'
inv: self.secondName = 'Tolstoy'
```

Инварианты в приведенном примере означают, что для любого объекта класса Book неизменными должны оставаться имя автора (значение поля firstName) и его фамилия (значение поля secondName). Строковые константы в OCL записываются в

обычных апострофах, а не в двойных, как это принято в языке программирования Java. Если класс ограничен несколькими инвариантами, то они «действуют совместно» и могут быть представлены одним булевым выражением в виде *конъюнкции инвариантов*. Поэтому последнее ограничение может быть переписано следующим образом:

```
context Book
inv: self.firstName = 'Lev' and self.secondName = 'Tolstoy'
```

В OCL операция конъюнкции записывается в виде служебного слова `and`.

Инвариант может иметь имя. Тогда оно указывается сразу же после служебного слова `inv`. Например,

```
context Book
inv author: self.firstName = 'Lev' and self.secondName = 'Tolstoy'
```

В приведенном примере инвариант имеет имя `author` (автор).

Другим видом OCL-ограничений, которые используются для уточнения класса, являются ограничения, при помощи которых можно вводить в модель класса *новые производные поля и новые методы-запросы к атрибутивной модели класса*.

Напомним, что значения базовых полей изменяются независимо друг от друга, а значения производных полей формируются из значений базовых полей. Например, базовым полем класса `Person` (личность) может быть поле `yearOfBirth` (год рождения), а производным — поле `age` (возраст). Ограничения, определяющие новые производные поля класса, имеют следующую структуру.

```
context <имя класса>
def:    <имя поля>:<тип поля> =
        <правило формирования производного поля>
```

В упомянутом выше классе `Person` базовым полем может быть также поле `firstName` (имя), а производным — поле `initial` (инициал). При помощи OCL-ограничения мы можем ввести в модель производное поле `initial`, хранящее первую букву имени, следующим образом:

```
context Person
def: initial:String = firstName.substring(1,1)
- -  initial - это первый символ имени.
```

Первое предложение приведенного примера специфицирует контекст производного поля, которым, в нашем случае, является класс `Person`. Второе предложение начинается со служебного слова `def`, за которым следует двоеточие. Это предложение задаёт имя и тип производного поля, а также *правило его формирования* в

виде OCL-выражения. Правило формирования производного поля является обязательным элементом def-предложения и отделяется от имени и типа производного поля при помощи символа равенства. Правило формирования производного поля в приведенном примере утверждает, что значение поля `initial` (инициал) формируется путём выделения первого символа из значения базового поля `firstName` (имя) при помощи операции `substring`.

Простейшие запросы к атрибутивной модели класса представляют собой стандартные get-методы, возвращающие значения соответствующих полей класса. В более сложном случае get-метод, реализующий запрос, может сформировать возвращаемое значение в виде функции, аргументами которой являются несколько атрибутов класса.

Стандартные get-методы вводятся в класс для реализации идеи инкапсуляции и обеспечения контролируемого доступа к его полям. Однако, в общем случае, get-методы могут рассматриваться как средство реализации произвольных запросов к атрибутивной модели класса. Поэтому язык объектных ограничений *OCL является не только средством для записи ограничений в формализованной и строгой форме, но и средством формулировки запросов к программной системе, представленной в виде UML-модели*.

Ограничения, определяющие новые методы для реализации запросов к атрибутивной модели, имеют следующую структуру:

```
context <имя класса>
def:    <заголовок метода-запроса> =
        <способ формирования возвращаемого значения>
```

В def-предложении записывается заголовок метода-запроса, включая тип возвращаемого значения и способ формирования возвращаемого значения. Как правило, имя такого метода начинается со слова «get». Способ формирования возвращаемого значения, в общем случае, может включать имя возвращаемого значения и OCL-выражение, специфицирующее то, каким образом оно вычисляется. Эта часть является обязательным элементом def-предложения и записывается справа от символа равенства. Например, простой запрос на основе стандартного метода `getLocation` (получить местонахождение) может быть введен в модель класса `Dwarf` при помощи OCL-ограничения следующим образом:

```
context Dwarf
def:    getLocation():Square = location
```

Первое предложение этого ограничения специфицирует контекст get-метода — класс `Dwarf`. Второе предложение, начинающееся со служебного слова `def`, определяет заголовок get-метода, а также способ формирования значения, возвращаемого этим методом. В нашем случае справа от символа равенства указано только имя поля `location` (местонахождение), значение которого возвращает метод `getLocation()`.

Рассмотрим пример более сложного запроса. Пусть имеется класс `Room` (комната), атрибутивная модель которого включает поля, определяющие размеры комнаты, а также местоположение и размеры окон и дверей. К атрибутивной модели этого класса можно сформировать запрос, возвращающий общую площадь стен комнаты с учетом оконных и дверных проемом. Такой запрос может быть реализован методом `getWallsArea` и специфицирован при помощи следующего OCL-ограничения:

```
context Room
def:    getWallsArea:float =
-- способ вычисления площади стен с учетом окон и дверей
```

Наличие в модели класса `Room` метода `getWallsArea` не означает, что его атрибутивная модель должна включать производное поле `wallsArea` для хранения значения площади стен, как функции значений базовых полей. Такое производное поле может отсутствовать.

### 10.2.2. Ограничение полей

Для уточнения полей используются следующие OCL-ограничения:

- ограничения, специфицирующие начальные значения полей;
- ограничения, специфицирующие правила формирования производных полей.

Специфицирование начальных значений полей необходимо для кодирования тех членов класса, при помощи которых осуществляется начальная инициализация полей (например, методов-конструкторов). Начальные значения полей, в простейших случаях, могут быть специфицированы в графическом символе класса, однако это можно сделать более точно при помощи ограничений, записанных на OCL. Ограничения, специфицирующие начальные значения полей, имеют следующую структуру:

```
context <имя класса>::<имя поля>:<тип поля>
init:   <OCL-выражение, задающее начальное значение поля>
```

В ограничениях, специфицирующих начальные значения полей, первое предложение описывает контекст в виде имени класса, а также имени и типа одного из полей этого класса. Между именем класса и именем поля размещается разделитель в виде двух символов «двоеточие». Второе предложение определяет начальное значение поля в виде OCL-выражения и начинается со служебного слова `init`. OCL-выражение может представлять собой константу того типа, под которым поле объявлено в классе. Разделителем между служебным словом `init` и его начальным значением является двоеточие.

Рассмотрим пример OCL-ограничения для специфицирования начального значения известного нам поля `depth` (глубина) класса `Submarine` (подводная лодка).

```
context Submarine::depth:float
init:   0.0
- - исходное положение подводной лодки – на поверхности
```

В приведенном ограничении контекст задаётся в виде имени класса `Submarine` и имени и типа поля `depth:float`, а начальное значение поля `depth` — в виде константы `0.0`. Ограничение уточняет исходную модель и означает, что когда в системе появляется новая подводная лодка (создаётся новый объект класса `Submarine`), то она должна находиться на поверхности.

Ниже приведены ещё несколько примеров для специфицирования начальных значений поля `cellPhone` (сотовый телефон) класса `Person` (личность) и поля `validity` (быть действительным) класса `CreditCard` (кредитная карта).

```
context Person::cellPhone:String
init:   '067-188-2633'

context CreditCard::validity:boolean
init:   true
```

Если в UML-модель некоторого класса введено производное поле, то оно должно быть уточнено при помощи нескольких ограничений. Во-первых, необходимо защитить это поле от несанкционированного изменения. Это делается путём введения естественно-языкового ограничения `{readOnly}`, которое означает, что данное поле предназначено только для чтения. Ограничение `{readOnly}` записывается непосредственно в графическом символе класса. Например,

```
area:float {readOnly}
```

Во-вторых, производное поле должно быть снабжено OCL-ограничением, специфицирующим правило его формирования. Ограничение, специфицирующее правило формирования *производного* поля, имеет следующую структуру:

```
context <имя класса>::<имя производного поля>:<тип производного поля>
derive: <правило формирования производного поля>
```

Ниже приведен пример OCL-ограничения для специфицирования правила получения производного поля `area` (площадь) класса `Rectangle` (прямоугольник):

```
context Rectangle::area:float
derive: width*height
```

В приведенном примере первое предложение задаёт контекст ограничения, а второе — правило формирования производного поля. Правило утверждает, что значение производного поля `area` (площадь) формируется путём умножения значения базового поля `width` (ширина) на значение базового поля `height` (высота).

Ниже приведен ещё один пример ограничения для специфицирования правила формирования производного поля `printName` (имя клиента, используемое при печати документов) класса `Customer` (заказчик).

```
context Customer::printName:String
derive: concat(title).concat(' ').concat(secondName)
```

OCL-выражение, использованное для записи правила формирования производного поля, означает, что производное поле `printName` формируется путём операции конкатенации значения поля `title` (титул), символа «пробел» и значения поля `secondName` (фамилия). OCL-операция конкатенации строк аналогична операции конкатенации, принятой в языке программирования Java и означает соединение нескольких строк в одну путём их последовательной записи без разделительных символов. Таким образом, предполагается, что при печати документов клиент именуется, например, в виде: г-н. Петров.

### 10.2.3. Ограничение методов

При уточнении методов используются следующие OCL-ограничения:

- ограничения, специфицирующие предусловия и постусловия методов;
- ограничения, специфицирующие методы-запросы к атрибутивной модели.

*Предусловие* метода — это условие, разрешающее или запрещающее его выполнение. Например, для того, чтобы выполнился метод, осуществляющий регистрацию водительских прав, необходимо, чтобы возраст владельца этих прав был равным или превышал восемнадцать лет. *В OCL предусловие формулируется в виде булевого OCL-выражения, которое проверяется перед выполнением метода*. Предусловие должно быть истинным в тот момент, когда метод начинает выполняться. Если предусловие принимает ложное значение, то метод не выполняется. Так же, как и в случае инварианта, программист может использовать предусловия для организации проверки правильности функционирования программы. Контролю в этом случае подвергаются условия выполнения метода. Если проверка показывает, что предусловие нарушается (принимает ложное значение), то это может рассматриваться как наличие исключительной ситуации, препятствующей выполнению метода.

*Постусловие* метода специфицирует изменения в системе, вызванные работой этого метода. Например, после того как выполнился метод регистрации водительских прав, в базе данных должна быть сформирована соответствующая запись, а владелец авторских прав должен получить счёт на оплату регистрации. *В OCL*

*постусловие формулируется в виде булевого OCL-выражения, которое проверяется после выполнения метода*. Постусловие должно быть истинным, если метод выполнился безошибочно. Если постусловие принимает ложное значение, то это означает, что метод выполнился с ошибкой. Постусловие также может использоваться для контроля правильности функционирования программы путём отслеживания моментов появления исключительных ситуаций.

В OCL при записи и инварианта, и предусловия, и постусловия используются булевы выражения, однако они различным образом контролируют правильность функционирования программы. В нормально функционирующей системе инвариант должен быть всегда истинным, а предусловие и постусловие должны быть истинными только в определённые моменты времени: до и после выполнения метода соответственно.

Инварианты, предусловия и постусловия образуют контракт класса, который может рассматриваться как набор ограничений, записываемых в виде булевых выражений и позволяющих разработчику и пользователю класса разделять одни и те же знания относительно поведения объектов этого класса. Исключительная ситуация возникает каждый раз, когда нарушается контракт класса.

Ниже приведен общий вид OCL-ограничения, специфицирующего предусловия и постусловия.

```
context <имя класса>::<заголовок метода>
pre <необязательное имя предусловия>:
    <булево OCL-выражение, которое должно быть истинным в момент
    начала выполнения метода>
post <необязательное имя постусловия>:
     <булево OCL-выражение, которое должно быть истинным после
     завершения выполнения метода>
```

Контекстом ограничения является метод, подлежащий ограничению при помощи предусловий и постусловий, с указанием имени класса, в котором он объявлен.

В некоторых случаях одно из условий (предусловие или постусловие) может отсутствовать. Если, например, метод должен безусловно выполняться, то в ограничении можно не специфицировать предусловие. Ниже приведен пример ограничения, в котором отсутствует предусловие, и, следовательно, метод может начинать выполняться безусловно. Ограничение специфицирует метод `setPrice` (установить цену) класса `Book` (книга).

```
context Book::setPrice(newPrice:Money):boolean
pre:    - - отсутствует
post:   price = newPrice
```

Смысл постусловия в приведенном примере в том, что после выполнения метода поле `price` (цена) класса `Book` должно содержать значение параметра `newPrice`

(новая цена) метода setPrice (установить цену). Отсутствие предусловия или постусловия в ограничении следует явно указывать при помощи комментария, а не исключать соответствующее предложение из ограничения. Если в ограничении отсутствует какое-либо предложение, например, pre-предложение, то это может быть расценено как синтаксическая ошибка, являющаяся следствием того, что разработчик модели просто забыл включить предложение в ограничение.

Безусловное выполнение метода можно указать и другим способом. Например, путем записи всегда истинного булевого выражения в предусловии. Ниже приведен пример OCL-ограничения, полученного из предыдущего примера, в котором предусловие означает безусловное выполнение метода.

```
context Book::setPrice(newPrice:Money):boolean
pre:    true
post:   price = newPrice
```

Иногда в булевом выражении post-предложения необходимо использовать значение одного и того же поля, но в разные моменты времени: до выполнения метода и после выполнения метода, специфицированного в контексте. Например, после выполнения метода birthday (день рождения) истинным должно быть булево выражение, означающее, что новое значение поля age (возраст) на единицу больше, чем предыдущее значение этого же поля. Для обозначения того факта, что в булевом выражении post-предложения необходимо использовать значение некоторого поля до выполнения метода, специфицированного в контексте, имя этого поля снабжается суффиксом @pre. Например, age@pre означает, что необходимо использовать значение поля age до выполнения метода birthday. Например,

```
context Person::birthday():void
pre:    true
post:   age = age@pre + 1
```

Специфицированный в контексте метод birthday класса Person (личность) увеличивает на единицу значение поля age. Предусловие разрешает безусловное выполнение метода birthday. Постусловие требует, чтобы после выполнения метода birthday новое значение поля age было на единицу больше, чем значение этого же поля, но до выполнения метода birthday.

Суффиксом @pre могут снабжаться не только имена полей, но и имена get-методов. Это означает, что в булевом выражении необходимо использовать возвращаемое значение этого метода, полученное до выполнения метода, специфицированного в контексте. Например,

```
context Dwarf::setLocation(newLocation:Square):void
pre:    true
post:   getLocation@pre():Square < > getLocation():Square
```

В приведенном примере ограничению подвергается метод `setLocation` класса `Dwarf` (см. подраздел 9.1.1). Предусловие разрешает безусловное выполнение этого метода, а смысл постусловия в том, что номер квадрата, куда перемещается гном после выполнения метода `setLocation`, не должен совпадать с номером квадрата, в котором он находился до выполнения метода `setLocation`.

Если в UML-модели класса уже имеется метод-запрос к его атрибутивной модели, то при помощи OCL-ограничения можно уточнить способ формирования возвращаемого значения для этого метода. В контексте ограничения указывается ограничиваемый метод и класс, в котором он определён, а возвращаемое значение специфицируется при помощи body-предложения. Таким образом, ограничения, специфицирующие возвращаемое значение метода-запроса, имеют следующую структуру:

```
context <имя класса>::<заголовок метода-запроса>
body:   <способ формирования возвращаемого значения>
```

Способ формирования возвращаемого значения записывается в виде OCL-выражения. Проиллюстрируем использование этого ограничения тем же примером, который мы использовали ранее для иллюстрации ввода в модель новых методов-запросов.

```
context Dwarf::getLocation():Square
body:   location
```

Первое предложение этого ограничения идентифицирует метод-запрос в виде стандартного метода `getLocation` (получить местонахождение) в контексте класса `Dwarf` (гном), а второе предложение, начинающееся со служебного слова `body`, определяет возвращаемое значение в виде значения поля `location` (местонахождение).

## 10.3. Базовые предопределенные типы данных в OCL

Одним из условий записи точных и полных OCL-ограничений является умение записывать синтаксически правильные OCL-выражения. Каждая переменная в OCL-выражении имеет тип, который определяет набор операций, допустимых для этой переменной.

Типы переменных в OCL делятся на предопределенные типы и типы, определяемые программистом. Предопределенные типы принято делить на *базовые типы* и *наборы однотипных данных*.

К базовым предопределенным типам данных относятся типы, описывающие: (1) данные булевого типа, (2) целые числа и вещественные числа и (3) строки символов. При записи OCL-ограничений в качестве имен базовых предопределенных

типов будем использовать имена, принятые в языке программирования Java: `int` (для целых чисел), `float` (для вещественных чисел), `String` (для строк символов) и `boolean` (для булевых данных).

Порядок выполнения операций над данными в сложных OCL-выражениях определяется их приоритетами, однако явное указание на порядок выполнения операций при помощи скобочной формы записи OCL-выражений является более предпочтительным, поскольку делает выражение яснее и нагляднее.

### 10.3.1. Булевы типы данных в OCL

Данные булевого типа могут принимать только два значения: `true` и `false`. Операции, допустимые для булевых типов данных в OCL, приведены в таблице на рис. 10.4.

| Наименование операции | Операция в OCL выражении | Тип результата операции |
|---|---|---|
| или | `a or b` | `boolean` |
| и | `a and b` | `boolean` |
| исключающее или | `a xor b` | `boolean` |
| отрицание | `not a` | `boolean` |
| равенство | `a = b` | `boolean` |
| неравенство | `a <> b` | `boolean` |
| следствие | `a implies b` | `boolean` |
| if-then-else | `if <булево OCL-выражение>`<br>`then <OCL-выражение>`<br>`else <OCL-выражение>`<br>`endif` | соответствует типу OCL-выражения в then- или else-предложении |

**Рис. 10.4.** Операции OCL для данных булевого типа

Операция «или» выполняется над двумя операндами булевого типа. Результат операции «или» принимает значение, равное `false`, только в том случае, когда оба операнда принимают значение `false`. При всех других комбинациях значений операндов результат операции «или» принимает значение `true`. Например, если некоторый класс содержит поля с именами `firstName` (имя) и `secondName` (фамилия) типа `String`, то можно составить следующее OCL-выражение булевского типа с использованием значений этих полей и операции «или»:

```
(firstName = 'Лев') or (secondName = 'Толстой')
```

Приведенное OCL-выражение будет принимать значение `true` в одном из трех случаев: (1) в поле `firstName` находится значение `'Лев'`; (2) в поле `secondName` находится значение `'Толстой'`; (3) в поле `firstName` находится значение `'Лев'`, а в поле `secondName` находится значение `'Толстой'`.

Операция «и» выполняется над двумя операндами булевого типа. Результат операции «и» принимает значение, равное `true`, только в том случае, когда оба операнда принимают значение `true`. При всех других комбинациях значений операндов результат операции «и» принимает значение `false`. Рассмотрим OCL-выражение, полученное из предыдущего OCL-выражения с заменой операции «или» на операцию «и».

```
(firstName = 'Лев') and (secondName = 'Толстой')
```

Приведенное OCL-выражение будет принимать значение `true` только в одном случае, когда в поле `firstName` находится значение `'Лев'`, а в поле `secondName` — значение `'Толстой'`.

Операция «исключающее или» выполняется над двумя операндами булевого типа и, в некотором смысле, является частным случаем операции «или». Результат операции «исключающее или» принимает значение, равное `true`, только в том случае, когда хотябы один из операндов, (но не оба), принимает значение `true`. Рассмотрим OCL-выражение

```
(firstName = 'Лев') xor (secondName = 'Толстой')
```

Теперь OCL-выражение будет принимать значение `true` в одном из двух случаев: (1) в поле `firstName` находится значение `'Лев'`; (2) в поле `secondName` находится значение `'Толстой'`.

Операция «отрицание» выполняется над одним операндом булевого типа и изменяет его значение на противоположное. Например, если некоторый класс содержит поле `lightOnOff` (свет включен/выключен) типа `boolean` и значение этого поля равно `true` (свет включен), то после применения операции

```
not lightOnOff
```

значение поля `lightOnOff` станет равным `false` (свет выключен).

Операция «неравенство» выполняется над двумя операндами, и ее результат принимает значение `true`, если операнды не равны, и значение `false`, если операнды равны. Например,

```
getAge() <> 18
```

В примере используется возвращаемое значение метода `getAge` (получить значение возраста). Если этот метод возвращает значение не равное 18, то приведенное OCL-выражение принимает значение `true`. Если же метод `getAge` возвращает значение, равное 18, то приведенное OCL-выражение принимает значение `false`.

Операция «равенство» выполняется над двумя операндами и, в некотором смысле, противоположна операции «неравенство». Результат операции «равенство» принимает значение `true`, если оба операнды равны, и значение `false`, если операнды не равны. Например, выражение

```
getAge() = 18
```

принимает значение `true`, если метод `getAge` возвращает значение, равное 18. Это же выражение принимает значение `false`, если метод `getAge` возвращает значение, не равное 18.

Операция «следствие» выполняется над двумя операндами булевого типа и является сокращенной формой операции «или», в которой первый операнд подвергается операции отрицания. Таким образом, выражение

`a implies b` эквивалентно выражению `(not a) or b`

Истинность результата операции «следствие» зависит от истинности первого операнда и может определяться при помощи следующих двух правил: (1) если значение первого операнда `false`, то значение всего OCL-выражения всегда `true`, вне зависимости от значения второго операнда; (2) если значение первого операнда `true`, то значениие всего OCL-выражения равно значению второго операнда (если второй операнд принимает значение `true`, то и все OCL-выражение принимает значение `true`, а если второй операнд принимает значение `false`, то и все OCL-выражение принимает значение `false`). Рассмотрим пример следующего OCL-выражения:

```
(getScore() < 60) implies test
```

В приведенном примере метод `getScore` (получить баллы) возвращает целое положительное число, соответствующее количеству баллов, полученных студентом в течение семестра по какой-либо из дисциплин, а поле `test` (зачет) типа `boolean` принимает значение `true`, если студент получил зачет по этой дисциплине, и значение `false` в противном случае. Если первый операнд (`getScore() < 60`) принимает значение `false`, то вне зависимости от значения поля `test`, OCL-выражение принимает значение `true`. Если же первый операнд принимает значение `true`, то значение OCL-выражения равно значению поля `test`.

Результатом операции «if-then-else» является одно из OCL-выражений записанных после служебных слов `then` или `else`, в зависимости от значения булевого выражения, записанного после служебного слова `if`. Синтаксис языка OCL не допускает сокращенной формы записи этой операции, в которой опущено предложение со служебным словом `else`. Например,

```
if getScore() > 60
   then test = true
   else test = false
endif
```

В приведенном примере метод `getScore` и поле `test` имеют такой же смысл, как и в предыдущем.

### 10.3.2. Типы данных OCL для представления чисел

Поскольку OCL является языком моделирования, а не языком программирования, и OCL-предложения не предназначены для трансляции в команды процессора, в OCL отсутствуют ограничения на диапазон представления чисел и отсутствует, например, такое понятие, как наибольшее допустимое целое число либо наибольшее допустимое число с плавающей запятой. Числа в OCL понимаются точно так же, как в математике. Операции для числовых данных, допустимые в OCL, приведены в таблице на рис. 10.5.

| Наименование операции | Операция в OCL-выражении | Тип результата операции |
|---|---|---|
| равенство | `a = b` | `boolean` |
| неравенство | `a <> b` | `boolean` |
| меньше | `a < b` | `boolean` |
| больше | `a > b` | `boolean` |
| меньше или равно | `a <= b` | `boolean` |
| больше или равно | `a >= b` | `boolean` |
| сложение | `a + b` | `int` или `float` |
| вычитание | `a - b` | `int` или `float` |
| умножение | `a * b` | `int` или `float` |
| деление | `a / b` | `float` |
| остаток от деления | `a.mod(b)` | `int` |
| целочисленное деление | `a.div(b)` | `int` |
| абсолютная величина | `a.abs()` | `int` или `float` |
| наибольшее | `a.max(b)` | `int` или `float` |
| наименьшее | `a.min(b)` | `int` или `float` |
| округление к ближайшему целому | `a.round()` | `int` |
| округление к наименьшему целому | `a.floor()` | `int` |

**Рис. 10.5.** Операции OCL для числовых данных

Операции «сложение», «вычитание», «умножение» и «деление» представляют собой обычные арифметические операции, и их свойства ничем не отличаются от свойств арифметических операций сложения, вычитания, умножения и деления.

Операции «равенство», «неравенство», «меньше», «больше», «меньше или равно» и «больше или равно» являются операциями сравнения. Операнды этих операций представляют собой целые либо вещественные числа, а результат — значение типа `boolean`. Поэтому, операции сравнения могут использоваться при записи булевых OCL-выражений в инвариантах, пред- и постусловиях.

Ниже приведено несколько примеров записи OCL-выражений с использованием операций сравнений.

```
200 * 2.2 + 100 = 540      выражение принимает значение true
200 * 2.2 + 100 <> 540     выражение принимает значение false
12 > 22.7                  выражение принимает значение false
12 > 22.7 = false          выражение принимает значение true
```

Операция «остаток от деления» позволяет получить остаток от деления двух целых чисел. Например, остатком от деления числа 13 на число 2 является число 1.

```
13.mod(2) = 1
```

Операция «целочисленное деление» позволяет получить частное от деления двух целых чисел в виде целого числа. Остаток от деления отбрасывается. Например, целая часть частного от деления числа 13 на число 2 является число 6.

```
13.div(2) = 6
```

Операция «абсолютная величина» позволяет получить значение целого или вещественного числа без знаков «+» или «-». Например,

```
-13.abs() = 13
```

Операции «наибольшее» и «наименьшее» позволяют выбрать наибольшее либо наименьшее число из двух целых или вещественных чисел. Например,

```
43.max(10) = 43
55.5.min(10) = 10
```

Операции «округление к ближайшему целому» и «округление к наименьшему целому» позволяют заменить вещественное число одним из ближайших целых чисел. Например,

```
(4.6).round() = 5
(4.4).round() = 4
(4.6).floor() = 4
(-2.6).floor() = -3
```

### 10.3.3. Строковые типы данных в OCL

При помощи строковых типов данных в OCL-выражениях представляются строки символов. Строковые литералы в OCL, в отличие от языка программирования Java, заключаются не в двойные кавычки, а в апострофы. Например, `'Андрей'` или `'Одесса'`. Операции для строковых типов данных, допустимые в OCL, приведены в таблице на рис. 10.6.

| Наименование операции | Операция в OCL-выражении | Тип результата операции |
|---|---|---|
| сцепление | `string.concat(string)` | `String` |
| размер | `string.size()` | `int` |
| верхний регистр | `string.toUpper()` | `String` |
| нижний регистр | `string.toLower()` | `String` |
| подстрока | `string.substring(int,int)` | `String` |
| равенство | `string1 = string2` | `boolean` |
| неравенство | `string1 <> string2` | `boolean` |

**Рис. 10.6.** Операции OCL для строковых типов данных

Операция «сцепление», называемая также операцией конкатенации, позволяет получить одну строку из двух строк путем их слияния. Например, операция «сцепление» позволяет из двух строк `'душа питается '` и `'знаниями'` получить строку `'душа питается знаниями'`.

```
'душа питается '.concat('знаниями') = 'душа питается знаниями'
```

Операция «размер», позволяет определить количество символов в строке. Например,

```
'душа питается знаниями'.size() = 22
```

Операция «верхний регистр» позволяет заменить все буквы строки на прописные (большие) буквы, а операция «нижний регистр» — все буквы строки на строчные (малые) буквы. Например,

```
'Василий Иванович'.toUpper() = 'ВАСИЛИЙ ИВАНОВИЧ'
'Василий Иванович'.toLower() = 'василий иванович'
```

Операция «подстрока» позволяет выделить из исходной строки ее подстроку. Выделяемая подстрока указывается при помощи двух целых чисел. Первое число указывает на первый символ подстроки в исходной строке, а второе — на последний символ подстроки в исходной строке. Например.

```
'Василий Иванович'.substring(9,16) = 'Иванович'
```

Операции «равенство» и «неравенство» посимвольно сравнивают две строки. Операция «равенство» возвращает значение `true` в том случае, если обе строки равны, а операция «неравенство» — в том случае, если строки не равны. Например.

```
('Василий' = 'Иванович') = false
('Василий' <> 'Иванович') = true
```

## Упражнения для практических занятий

10.1. Для класса `Airplane`, моделирующего пассажирский самолёт, предложите три инварианта, ограничивающие количество продаваемых билетов, общий вес багажа и максимальное расстояние, пролетаемое самолетом без дозаправки. Сформулируйте их на естественном языке и в виде OCL-ограничений. Используйте все известные вам способы записи инвариантов.

10.2. Предложите несколько инвариантов для класса `BankAccount` (банковский счёт), ограничивающие минимальную сумму, хранящуюся на счете, и наибольшее количество владельцев счета. Сформулируйте предложенные инварианты на естественном языке и в виде OCL-ограничений.

10.3. Исследуйте применимость инвариантов для случая наследования. Можно ли инварианты, разработанные для суперкласса, использовать для ограничения подкласса? Проиллюстрируйте ваши выводы на примере системы, состоящей из одного суперкласса и одного подкласса.

10.4. Предложите список базовых полей для класса `Airplane`, моделирующего пассажирский самолёт, и при помощи OCL-ограничений определите два новых производных поля этого класса. Какими ограничениями уточняются производные поля?

10.5. Предложите список полей для класса `Airplane`, моделирующего пассажирский самолёт, и при помощи OCL-ограничений определите два новых `get`-метода этого класса.

10.6. Предложите список базовых полей для класса `Student` (студент) и при помощи OCL-ограничений определите несколько новых производных полей этого класса.

10.7. Предложите базовые и производные поля, а также методы класса прямоугольных треугольников. Целью моделирования является представление школьных знаний о прямоугольных треугольниках средствами

UML-OCL. Предложите инварианты, ограничивающие объекты этого класса. Сформулируйте их на естественном языке и запишите в виде OCL-ограничений. Включите в модель OCL-ограничения для производных полей и get-методов.

10.8. Диаграмма классов на рис. 9.10, моделирует систему с единичным наследованием. Уточните классы, а также поля и методы на этой диаграмме при помощи известных вам OCL-ограничений. Используйте OCL-ограничения для определения новых полей и методов.

10.9. Разработайте атрибутивную модель для класса объектов солнечной системы. В качестве объектов Солнечной системы будем рассматривать Солнце, планеты и спутники планет. Запишите OCL-ограничения, определяющие объекты Земля и Луна.

10.10. Разработайте атрибутивную модель класса, моделирующего химические элементы в периодической системе элементов Менделеева. Запишите OCL-ограничения, определяющие объекты: водород и гелий.

10.11. Для класса `BankAccount` (банковский счёт) естественным является метод `withdraw` (снятие наличных со счёта). Запишите OCL-ограничения, специфицирующие пред- и постусловия для этого метода. Будем считать, что снятие наличных со счета разрешается только его владельцу в том случае, если он не имеет большой задолженности по кредиту. После выполнения операции `withdraw` на банковском счете всегда должна оставаться минимально допустимая сумма.

10.12. Пусть класс, моделирующий автоматическую стиральную машину, включает метод `heater` (нагреватель). Запишите OCL-ограничения, специфицирующие пред- и постусловия для этого метода.

10.13. Предложите структуру OCL-ограничений для специфицирования пред- и постусловий стандартных get- и set-методов. Для записи структуры OCL-ограничений используйте нотацию подрадела 2.2.

10.14. Предложите модель класса `Employee` для моделирования наёмных работников некоторого предприятия. Пусть в графическом символе класса атрибутивная модель включает только фамилию работника, его заработную плату и стаж работы, а поведение моделируется единственным методом, позволяющим установить новую заработную плату. Уточните графический символ класса при помощи всех известных вам ограничений.

Раздел 11

# МОДЕЛИРОВАНИЕ КЛАССОВ И ИНТЕРФЕЙСОВ

Классы являются основными «строительными блоками», из которых конструируется модель пространственной структуры системы. В настоящем разделе рассматриваются те средства UML и OCL, которые используются для представления этих «строительных блоков». Если рассматривать содержание настоящего раздела с точки зрения моделирования структуры системы, то он посвящён изучению моделирования простейшей системы, состоящей только из одного класса. Часто класс конструируется путём реализации одного или нескольких интерфейсов. Поэтому в разделе рассматривается также и UML-нотации для интерфейсов.

## 11.1. Графические символы класса

Ранее, при изучении идеи наследования, мы познакомились с графическим символом класса, представляющим собой прямоугольник, разделённый на три отделения горизонтальными линиями. Это наиболее полная форма изображения класса на UML диаграммах. Она приведена на рис. 11.1.

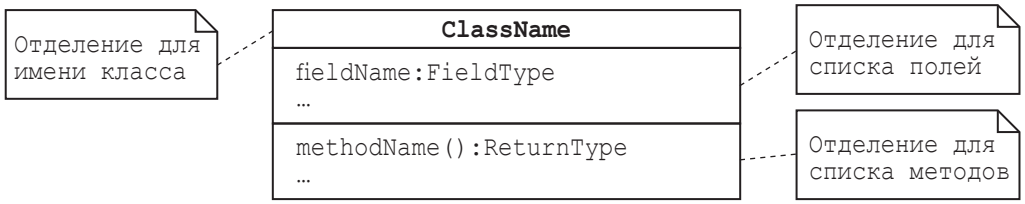



**Рис. 11.1.** Графический символ класса в полной форме

Верхнее отделение графического символа класса используется для записи имени класса.

Среднее отделение предназначено для специфицирования полей класса. Описание одного поля должно занимать точно одну строку. Поэтому в конце строки

не используются какие-либо разделители. Описание отдельного поля, в простейшем случае (без учёта ограничения), состоит из двух частей: имени поля и типа поля. Обе части разделяются двоеточием. При записи имени типа поля, а также типов входных параметров и возвращаемых значений методов будем использовать имена, принятые в языке программирования Java.

Нижнее отделение предназначено для описателей методов класса. Описание одного метода размещается в одной строке. В простейшем случае (без учёта ограничений) при описании метода записывается следующая последовательность: имя метода, перечень входных параметров в круглых скобках, двоеточие, тип возвращаемого значения. Разработчики модели класса, как правило, оперируют с методами, возвращающими одно значение. Однако UML позволяет описывать методы, возвращающие несколько значений. Если входные параметры отсутствуют, то круглые скобки остаются пустыми. Если метод ничего не возвращает, то, вместо типа возвращаемого значения записывается служебное слово `void` (пусто). Каждый входной параметр метода записывается по тем же правилам, что и поле: имя входного параметра и тип входного параметра.

Имена классов, полей, методов или входных аргументов записываются на английском языке. Имя класса всегда начинается с прописной (большой) буквы и записывается жирным шрифтом. Имя поля, метода или входного параметра всегда начинается со строчной (маленькой) буквы и записывается нежирным шрифтом. Имена могут быть составными (составленными из нескольких слов). В этом случае имя записывается без пробелов между словами, а каждое новое слово начинается с прописной (большой) буквой. Пример правильно записанного составного имени класса — `SavingAccount` (сберегательный счёт), а пример правильного записанного составного имени поля — `dateOfBirth` (дата рождения).

Существует, также, несколько сокращённых форм графического символа класса, приведенных на рис. 11.2.

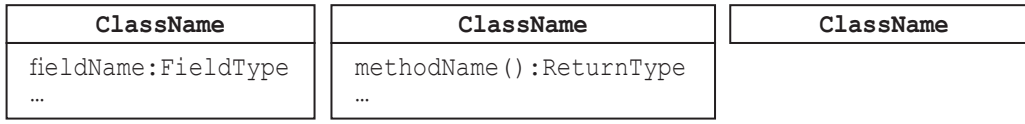



**Рис. 11.2.** Графический символ класса в сокращённых формах

В некоторых случаях (например, при моделировании таблиц реляционных баз данных) класс может быть представлен исключительно атрибутивной моделью в виде набора полей. Тогда отделение методов может отсутствовать. В ряде случаев класс может быть представлен только своими методами (или даже одним методом). В этих случаях может отсутствовать отделение полей. Левый и средний графические символы класса на рис. 11.2. иллюстрируют отмеченные случаи. Символ класса, приведенный в правой части рис. 11.2, используется тогда, когда для целей моделирования необходимо обозначить присутствие класса, а его

детальная структура несущественна. Этот символ не означает, что существуют классы, состоящие только из своих имён.

Графический символ класса, а также описания его полей и методов могут быть уточнены при помощи ограничений.

### 11.1.1. Специфицирование класса при помощи стереотипов

При помощи ограничений можно уточнить модель класса, специфицировав его сферу применимости. В UML имеется ещё один механизм уточнения модели класса (а вообще говоря, любого элемента модели программной системы), называемый *стереотипом*. При моделировании класса стереотипы используются тогда, когда необходимо указать на принадлежность данного класса к некоторому *метаклассу*. Под метаклассом будем понимать *класс классов, или класс, объектами которого являются классы*. Стереотипы обычно используются в двух случаях. Во-первых, для уточнения модели, построенной с использованием базовых средств UML. Во-вторых, для создания *профиля* или *диалекта* языка, ориентированного на моделирование в рамках некоторой платформы программирования (например, J2EE или .NET) или сферы применимости (например, моделирование систем реального времени, Web-приложений или баз данных).

Стереотип имеет простой синтаксис и представляет собой слово на английском языке (имя стереотипа), заключённое в двойные угловые скобки. Вместо двойных угловых скобок можно использовать символы «меньше» и «больше» (<< >>). Имя стереотипа начинается с прописной (большой) буквы и является, по сути, именем метакласса, к которому принадлежит элемент языка, снабжённый стереотипом. Стереотип всегда записывается перед тем элементом, который он уточняет. При помощи стереотипов можно уточнять весь класс или его члены. В случае уточнения всего класса, стереотип записывается в верхнем отделении графического символа класса, непосредственно перед именем класса. На рис. 11.3 приведен пример графического символа класса, снабжённого стереотипом.

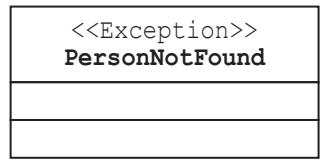



**Рис. 11.3.** Пример символа класса, снабжённого стереотипом

В примере на рис. 11.3 класс с именем `PersonNotFound` (личность не найдена) помечен стереотипом `<<Exception>>` (исключение). Стереотип `Exception` уточняет модель класса и предполагает использование языков программирования, в которых типизируются исключительные ситуации, а каждая исключительная ситуация является объектом некоторого класса. Стереотип `<<Exception>>` означает, что

существует некоторый *метакласс* исключительных ситуаций с именем `Exception`, а класс `PersonNotFound` является одним из классов этого метакласса. Разработчик модели может по своему усмотрению уточнять класс при помощи стереотипов в том смысле, что он имеет право придумывать свои собственные имена стереотипов. Такие стереотипы называются *стереотипами, определяемыми пользователем*. Стереотип `<<Exception>>` на рис. 11.3 является стереотипом, определяемым пользователем.

Однако, в официальной документации по UML, размещённой на Web-сайте OMG (Object Management Group), приводится список стандартных стереотипов, имена которых являются служебными словами, предопределёнными официальной документацией. В таблице на рис. 11.4 приведен список нескольких стандартных стереотипов, которые можно использовать для уточнения всего класса.

| Имя стереотипа | Назначение стереотипа |
|---|---|
| `<<Auxiliary>>` | Класс, специфицированный стереотипом `<<Auxiliary>>`, имеет статус вспомогательного по отношению к другому, более фундаментальный классу, специфицированному стереотипом `<<Focus>>`. |
| `<<Focus>>` | Класс, специфицированный стереотипом `<<Focus>>`, определяет основные свойства и поведение для одного или нескольких вспомогательных классов, специфицированных стереотипом `<<Auxiliary>>`. |
| `<<Type>>` | Класс, специфицированный стереотипом `<<Type>>`, определяет домен (область изменения) множества объектов без указания того, каким образом физически реализуются объекты. |
| `<<Primitive>>` | Класс, специфицированный стереотипом `<<Primitive>>`, определяет свойства и поведение одного из встроенных (примитивных) типов данных. |
| `<<Interface>>` | Класс, специфицированный стереотипом `<<Interface>>`, определяет свойства и поведение интерфейса. |

**Рис. 11.4.** Стандартные стереотипы, используемые для профилирования класса

На рис. 11.5. приведены примеры графических символов класса, принадлежащих метаклассу `Primitive`.

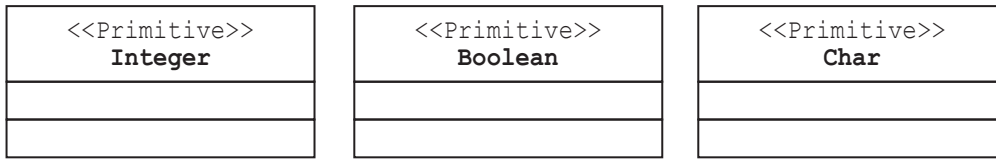



**Рис. 11.5.** Примеры классов, снабжённых стандартным стереотипом `<<Primitive>>`

В тех случаях, когда недостаточно указать только имя метакласса, а необходимо специфицировать его атрибут, стереотип снабжается *тегированным значением*. При помощи тегированных значений описываются свойства стереотипа, а поскольку стереотип рассматривается нами как метакласс, то тегированное значение можно рассматривать как метаполе (поле метакласса). На рис. 11.6 приведен пример графического символа класса, уточнённого при помощи стереотипа и снабжённого тегированным значением.

Тегированное значение размещается в графическом символе комментария. Вначале записывается имя стереотипа, а затем — само тегированное значение. Тегированное значение представляет собой выражение, состоящее из левой и правой частей, которые разделены символом равенства. В левой части записывается *тег*, а в правой — значение тега. В примере на рис. 11.6. при помощи тегированного значения уточнено, что метакласс `Exception` обладает свойством контролировать исключительные ситуации на этапе (`phase`) выполнения программы (`runtime`).

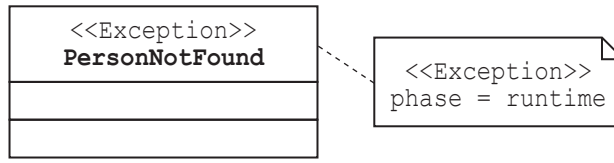



**Рис. 11.6.** Пример стереотипа, снабжённого тегированным значением

*Тегированное значение может быть использовано только после того, как некоторый элемент языка уточнён при помощи стереотипа*.

Стереотип может быть специфицирован при помощи нескольких тегированных значений.

## 11.2. Префиксы видимости полей и методов

Префиксы видимости представляют собой специальные символы, располагаемые в первой позиции строки, описывающей поле или метод. Префикс видимости указывает на уровень доступа к полю или методу. Наименование «префикс видимости» означает, что при помощи специального символа записывается информация о том, из каких частей объектной системы «видимо» (доступно) данное поле или метод. Префиксы видимости, совместно со стандартными get- и set-методами являются средством реализации идеи инкапсуляции.

UML использует четыре префикса видимости, обеспечивающие четыре уровня доступа к полям или методам класса.

- Общий или *public* префикс видимости (символ «+»). Если член класса помечен public префиксом видимости, то он доступен в контексте всей объектной программы.

- Защищённый или *protected* префикс видимости (символ «#»). Если член класса помечен protected префиксом видимости, то он доступен в контексте данного класса и всех его подклассов.
- Пакетный или *package* префикс видимости (символ «~»). Если член класса помечен package префиксом видимости, то он доступен в контексте пакета классов, содержащего данный класс.
- Секретный или *private* префикс видимости (символ «–»). Если член класса помечен private префиксом видимости, то он доступен только в контексте своего класса.

Таким образом, если рассматривать префиксы видимости как характеристики степени доступа к членам класса, то public префикс видимости обеспечивает наибольшую степень доступа, а затем, в порядке убывания, следуют protected, package и private.

Приведенные ниже рисунки иллюстрируют «работу» префиксов видимости для случая, когда префиксом помечается поле класса. На рис. 11.7 изображены классы: `A`, `B`, `C` и `D`. Классы `A` и `B` находятся в пакете с именем `Package`, а классы `C` и `D` — вне этого пакета. Более подробно пакеты будут изучаться в последующих разделах, а пока под пакетом будем понимать группу, объединяющую несколько классов. Класс `C` является подклассом класса `A`. В классе `A` описано поле `publicField`, снабжённое public префиксом видимости в виде символа «+» в первой позиции строки, а также некоторый метод `aMethod`. Жирные стрелки показывают, из каких элементов программы «видимо» поле `publicField` или из каких элементов программы разрешён доступ к этому полю.

Рисунок 11.7 позволяет легко расшифровать смысл public уровня доступа к членам класса.

Если поле класса помечено public префиксом видимости, то оно доступно:

- из методов данного класса (класс `A`);
- из методов подкласса данного класса (класс `C`);
- из методов классов, входящих в пакет, в котором находится данный класс (класс `B`);
- из методов других классов, не являющихся подклассами данного класса и не входящих в пакет, в котором находится данный класс (класс `D`).

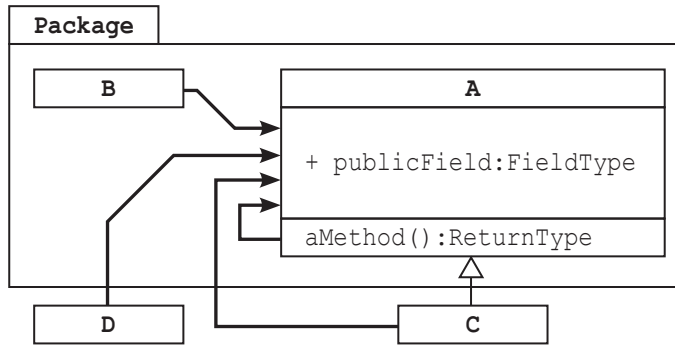



**Рис. 11.7.** Доступ к полю, помеченному `public` префиксом видимости

Таким образом, если все поля класса помечены public префиксом видимости, то класс полностью открыт для внешнего доступа, а его объекты не инкапсулированы. Использование public-полей в модели класса не соответствует объектно-ориентированной парадигме, и их следует объявлять лишь в тех случаях, когда влияние таких полей на остальную часть системы минимально. Например, когда поле представляет собой поименованную константу с `{readOnly}` ограничением.

На рис. 11.8 изображена та же система классов и пакетов, однако поле класса `A` (поле `protectedField`) снабжено префиксом видимости protected в виде символа «#» в первой позиции строки. Стрелки на рис. 11.8 показывают, из каких элементов программы «видимо» поле `protectedField`.

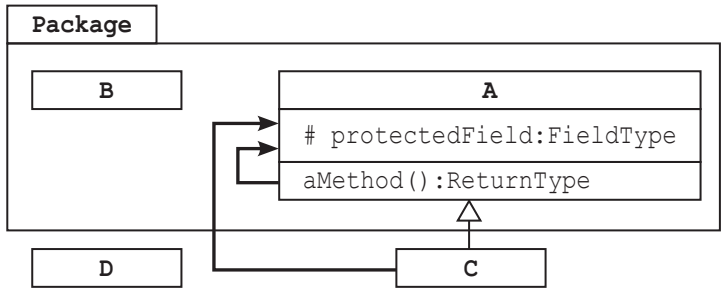



**Рис. 11.8.** Доступ к полю, помеченному `protected` префиксом видимости

Рис. 11.8 показывает, что если поле помечено `protected` префиксом видимости, то оно доступно только:

- из методов данного класса (класс `A`) и
- из методов подкласса данного класса (класс `C`).

Доступ к protected-полю из методов других классов, входящих в пакет, в котором находится данный класс, а также из методов классов, не входящих в пакет, в котором находится данный класс, невозможен. Использование protected префикса видимости как бы распространяет идею инкапсуляции на иерархию наследования. Поэтому protected префиксом видимости целесообразно снабжать те члены класса, к которым предполагается обращаться из его подклассов. В этом случае, например, обращение из подкласса к protected-полям суперкласса может быть непосредственным, без использования set- и get-методов.

На рис. 11.9 опять изображена та же система классов и пакетов, однако поле класса `A` снабжено префиксом видимости package в виде символа «~». Жирные стрелки показывают из каких элементов программы «видимо» поле `packageField`. Как следует из рис. 11.9, если поле помечено package префиксом видимости, то оно доступно только:

- из методов данного класса (класс `A`) и
- из методов классов, входящих в пакет, в котором находится данный класс (класс `B`).

Доступ к package-полю из методов подкласса, а также из методов классов, не входящих в пакет, в котором находится данный класс, невозможен.

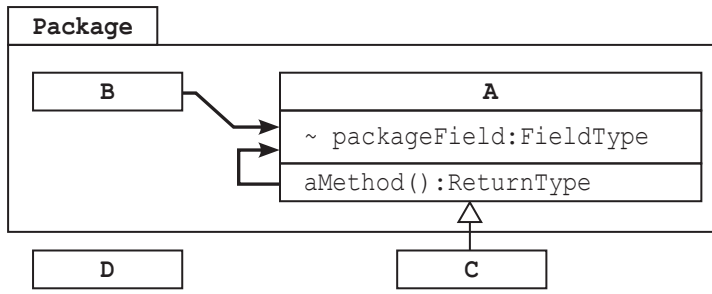



**Рис. 11.9.** Доступ к полю, помеченному `package` префиксом видимости

И, наконец, рис. 11.10 иллюстрирует «работу» private префикса видимости. Видно, что если поле снабжено private префиксом видимости, то оно доступно только методам своего класса.

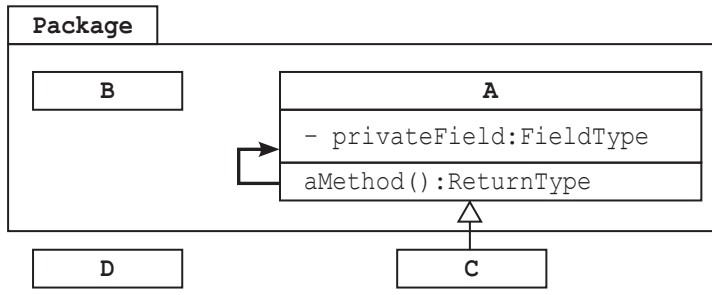



**Рис. 11.10.** Доступ к полю, помеченному `private` префиксом видимости

UML и язык программирования Java допускают произвольное использование префиксов видимости при описании полей и методов. Однако, реализация идеи инкапсуляции обеспечивается некоторыми правилами совместного использования префиксов видимости и стандартных set- и get-методов.

Для того, чтобы объекты, создаваемые при помощи некоторого класса, были инкапсулированы, а его поля полностью сокрыты, необходимо:

1. описать поля класса с *private* префиксами видимости;
2. снабдить поля соответствующими set- и get-методами;
3. описать методы класса с *public* префиксами видимости.

## 11.3. Описание полей

Поля моделируют атрибуты класса, а их совокупность представляет собой атрибутивную модель класса. Ранее мы познакомились с элементами классификации полей. Например, мы определили нестатические и статические поля, а также базовые и производные поля. Однако это не все виды полей, которые могут использоваться при моделировании класса. Более полный список включает следующие альтернативные виды:

- базовые или производные поля;
- поля с единичными или множественными значениями;
- нестатические или статические поля;
- поля, определяемые в классе или поля, определяемые ассоциацией.

Альтернативность видов полей в приведенном списке означает, что при классификации поля необходимо выбирать только один из его видов, записанных в строке. Например, поле не может быть одновременно базовым и производным либо нестатическим и статическим. В то же время характеристики, использованные в приведенной классификации, являются независимыми, поэтому, например, некоторое поле может быть одновременно и производным, и иметь множественные значения либо быть базовым, нестатическим, иметь единичное значение и определяться в классе. Приведенная классификация важна, поскольку знание о разнообразии видов полей даёт возможность разработчику создавать более точные и адекватные модели.

### 11.3.1. Базовые и производные поля

*Базовые поля моделируют независимые атрибуты класса*. Это означает, что значение какого-либо базового поля не зависит от значений других базовых полей.

После того, как при помощи класса создан конкретный объект и проведена начальная инициализация его полей, дальнейшая «судьба» базовых полей может быть различной. Некоторые базовые поля в процессе функционирования объекта изменяют свои значения. Например, поле `location` (местонахождение) в примере с гномом, рассмотренном в подразделе 9.1.1, изменяет своё значение при перемещении гнома.

Другие базовые поля должны сохранять значения, полученные при начальной инициализации на протяжении всего срока существования объекта. Более того, изменение значений таких полей недопустимо. Примером такого поля может быть поле `dateOfBirth`, хранящее дату рождения некоторой личности. Отмеченная специфика базовых полей отражается в их описании.

Базовые поля, значение которых должно оставаться неизменным в процессе функционирования объекта, снабжаются ограничением `{readOnly}`, которое означает, что значение этого поля можно только прочесть, но нельзя изменить путём записи в него нового значения. На рис. 11.11 приведена модель класса `Person` (личность), в которой описаны только базовые поля. Поле с ограничением `{readOnly}`

должно оставаться неизменными на протяжении всей «жизни» любого объекта этого класса.

В модели класса `Person`, приведенной на рис. 11.11, использованы три базовых поля. Значения полей `firstName` (имя) и `secondName` (фамилия) любого объекта этого класса могут изменяться, и, следовательно, значения этих полей разрешается не только читать, но и изменять путём записи в них новых значений. Значение поля `dateOfBirth` (дата рождения) после его начальной инициализации должно оставаться неизменным, и, следовательно, его значение разрешается только читать.

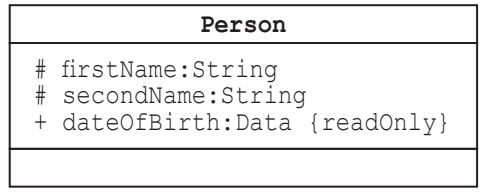



**Рис. 11.11.** Пример описания базовых полей

Поля `firstName` и `secondName` снабжены protected префиксом видимости. Это позволит методам подкласса класса `Person` обращаться к этим полям непосредственно по имени, а поле `dateOfBirth` снабжено public префиксом видимости, поскольку после создания объекта и начальной инициализации его полей поле `dateOfBirth` перестаёт быть переменной и превращается в поименованный литерал.

Существует несколько способов отображения *базовых полей*, *снабжённых ограничением* `{readOnly}`, в Java-код. Чаще всего такие поля рассматриваются как final-поля и снабжаются модификатором final. На рис. 11.12 приведен Java-код, соответствующий модели класса `Person`.

```
class Person {

   // описание полей
   protected String firstName;
   protected String secondName;
   public final Data dateOfBirth;

   // описание методов
           . . .
}
```

**Рис. 11.12.** Пример отображения базовых полей в Java-код

*Производные поля моделируют атрибуты класса, значения которых зависят от значений базовых полей*. Поэтому производные поля можно рассматривать как

функции базовых полей, и, следовательно, изменение значения базового поля должно сопровождаться изменением значения соответствующего производного поля.

Поскольку значения производных полей формируются из значений соответствующих базовых полей и не могут изменяться произвольным образом, их описание должно включать ограничение `{readOnly}`. Однако использование только ограничения `{readOnly}` при описании производных полей недостаточно, поскольку не позволяет отличить их от базовых полей, значения которых должны оставаться неизменными. Поэтому *описание производных полей должно быть обязательно дополнено OCL-ограничениями*, детерминирующими способ формирования значений производных полей при помощи derive- или body-предложений.

Ограничение `{readOnly}` используется как при описании базовых полей, так и при описании производных полей. Однако, отображение производных полей в Java-код отличается от отображения базовых полей с ограничением `{readOnly}`. Производные поля не могут быть снабжены модификатором final, который означает, что значение поля после его начальной инициализации ни при каких обстоятельствах нельзя изменить, поскольку значения производных полей изменяются синхронно с изменениями их базовых полей. *Для чтения значений производных полей используются get-методы*, возвращающие текущие значения производных полей, сформированные в соответствии с ограничениями, специфицирующими правила их формирования. Иными словами, зависимость значения производного поля от соответствующего базового поля реализуется кодом get-метода. На рис. 11.13 приведен пример модели класс `Circle` (окружность), в котором описаны базовое и производные поля, а также два get-метода, обеспечивающие чтение текущих значений производных полей.

Модель класса, приведенная на рис. 11.13, включает два OCL-ограничения с body-предложениями, при помощи которых специфицируются способы формирования значений производных полей в виде возвращаемых значений get-методов. В модели имеется одно базовое поле `radius` (радиус) и два производных поля: `circuit` (длина окружности) и `area` (площадь круга). При описании производных полей использованы как ограничения на естественном языке `{readOnly}`, размещённые в строках, описывающих производные поля, так и OCL-ограничения. При помощи body-предложений показано, каким образом осуществляется вычисление производных полей `circuit` и `area`.

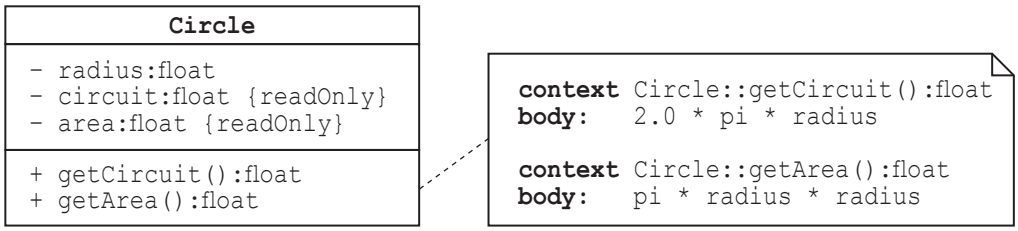



**Рис. 11.13.** Пример описания производных полей

При чтении модели, приведенной на рис. 11.13, наличие OCL-ограничений позволяет идентифицировать поля `circuit` и `area` не как базовые, а как производные.

На рис. 11.14 приведен Java-код, соответствующий модели класса `Circle`.

```
class Circle {

   // описание полей
   private float radius;
   private float circuit;
   private float area;
   public final float pi = 3.14;

   // описание методов
   public float getCircuit(){
      circuit = 2.0 * pi * radius;
      return circuit;
      {
   public float getArea(){
      area = pi * radius * radius;
      return area;
      {
}
```

**Рис. 11.14.** Отображение производных полей в Java-код

Как это следует из модели на рис. 11.13 и кода на рис. 11.14, класс `Circle` не содержит set-методов для полей `circuit` и `area`. Поэтому нет никакой возможности безусловно установить новые значения этих полей. Однако можно легко прочитать их значения при помощи get-методов, формирующих текущие значения производных полей из текущего значения базового поля `radius`.

### 11.3.2. Поля с множественными значениями

В рамках ООП нет необходимости в понятии данное. Любое данное может мыслиться как объект соответствующего класса. Поэтому любое поле класса может рассматриваться как объект. Например, поле `firstName` (имя) в классе `Person` (личность) является, по сути, указателем на объект класса `String`.

В рассмотренных выше примерах описания полей отдельное поле соответствовало только одному объекту. Однако, несложно найти пример, когда атрибут класса должен моделироваться полем, соответствующим множеству объектов. Например, поле `authors` (авторы) класса `Book` (книга) должно моделировать множество авторов книги.

UML позволяет включать в атрибутивную модель класса атрибуты, соответствующие множеству объектов, и поэтому поле класса может быть либо *полем с единичным значением*, либо *полем с множественными значениями*. Поле с единичным значением моделирует атрибут класса, соответствующий одному объекту, а поле с множественными значениями — атрибут, соответствующий множеству объектов. При описании поля с множественными значениями используется специальное выражение, специфицирующие *множественность* поля. Это выражение записывается в квадратных скобках сразу же за типом поля. Например, поле `authors` может быть описано следующим образом:

```
authors:String[1..5]
```

Приведенный пример означает, во-первых, что поле `authors` является множественным, а во-вторых, что количество объектов этого множества (количество авторов) может находиться в диапазоне от единицы до пяти. Ниже приведены выражения, которые могут использоваться для указания множественности полей.

`M..N` – от `M` до `N` (где, `M` и `N` целые положительные числа)
`0..1` – ноль или один
`1..1` – один и только один
`0..*` – от нуля до любого положительного целого
`1..*` – от единицы до любого положительного целого

Ясно, что поле с единичным значением является частным случаем поля с множественным значением, когда множественность поля специфицируется выражением [1..1]. Как правило, при описании поля с единичным значением это выражение опускают.

На рис. 11.15 приведен пример модели класса `Person`, атрибуты которого моделируются полями с единичными и множественными значениями.

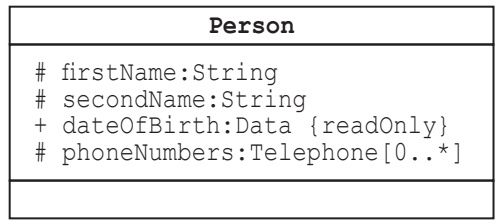



**Рис. 11.15.** Пример описания поля с множественными значениями

Модель класса `Person` на рис. 11.15 построена из модели, приведенной на рис. 11.11 путем добавления нового поля `phoneNumbers` (номера телефонов) с

множественными значениями. Это поле моделирует номера телефонов личности. Его множественность `[0..*]` означает, что личность может либо не иметь ни одного номера телефона, либо иметь некоторое количество номеров телефонов.

Ранее отмечалось, что UML позволяет включать в модель класса методы, возвращающие не одно, а множество значений. В этом случае возвращаемые значения метода специфицируется так же, как поля с множественными значениями. Например, стандартный get-метод, возвращающий множество значений номеров телефонов, моделируемых полем с множественными значениями `phoneNumbers`, может быть описан в виде

```
getPhoneNumb():Telephone[0..*]
```

При отображении полей с множественными значениями в программный код они рассматриваются как массивы элементов соответствующих типов. Так, например, поле `phoneNumbers` может быть отображено в одномерный массив элементов типа `Telephone` следующим образом:

```
Telephone[] phoneNumbers;
```

### 11.3.3. Статические поля

Статические поля описывают атрибуты класса как отдельной сущности. Значения этих атрибутов не зависят от значений атрибутов объектов, составляющих класс. Класс содержит только один комплект статических полей вне зависимости от того, сколько объектов создано при помощи этого класса. Количество статических полей класса постоянно. При моделировании класса статические поля включаются в общий список полей. Для того, чтобы описания нестатических полей отличались от описаний статических полей, описания последних подчёркиваются.

На рис. 11.16 приведена модель класса `Person` в которую включено описание статического поля `numbOfPersons` (количество личностей).

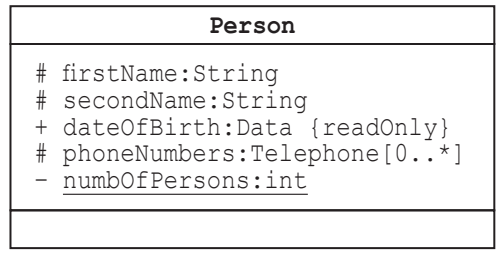



**Рис. 11.16.** Пример описания статического поля

Поле numbOfPersons на рис. 11.16 снабжено private префиксом видимости. Это означает, что к этому полю имеют доступ только методы класса Person, например, конструкторы этого класса.

При отображении описаний статических полей в программный код используется модификатор static. На рис. 11.17 приведен пример отображения модели класса Person, изображённой на рис. 11.16 в код.

```
class Person {

   // описание полей
   protected String firstName;
   protected String secondName;
   public final Data dateOfBirth;
   protected Telephone[] phoneNumbers;
   private static int numbOfPersons;

   // описание методов
            . . .
}
```

**Рис. 11.17.** Отображение статических полей в код

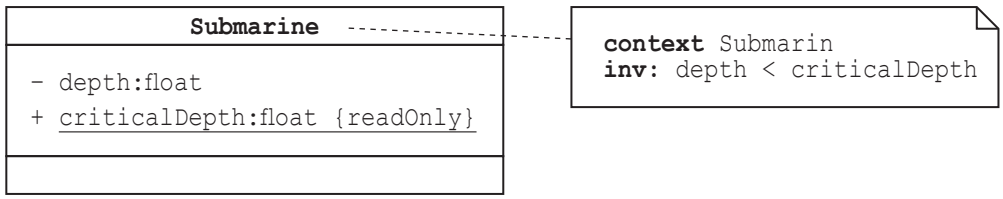
Инварианты класса близки по смыслу статическим полям, поскольку характеризуют всё множество объектов класса, а, следовательно, и весь класс как отдельную сущность. Поэтому, если модель класса включает OCL-ограничения, специфицирующие его инварианты, то часто одна или несколько переменных OCL-выражения, используемого в инварианте, являются статическими полями класса. Проиллюстрируем сказанное следующим примером. В подразделе 10.2.1 приведен пример ограничения класса подводных лодок Submarin при помощи инварианта следующим образом:

```
context Submarin
inv: depth < criticalDepth
```

Ограничение означает, что текущая глубина погружения любой подводной лодки класса Submarin, определяемая значением поля depth (глубина), не может превышать некоторую критическую глубину, определяемую значением поля criticalDepth. Значение критической глубины характеризует весь класс подводных лодок, и поэтому поле criticalDepth целесообразно описать как статическое.

Поле criticalDepth после его начальной инициализации не должно изменяться, и, следовательно, его необходимо снабдить {readOnly} ограничением. Такие поля часто называют *статическими литералами*. Статические литералы, как и

обычные поименованные литералы, предназначены только для чтения. и поэтому могут быть снабжены префиксом видимости `public`. На рис. 11.18 приведена модель класса `Submarin` и пример отображения этой модели в код.




```
class Submarine {

   // описание полей
   private float depth;
   public static final float criticalDepth;

   // описание методов
           . . .
}
```

**Рис. 11.18.** Модель класса с инвариантом, использующим статическое поле, и пример отображения этой модели в код

### 11.3.4. Поля, определяемые ассоциацией

Модель системы, представленная одним классом, является структурно простой, что, однако, не всегда означает, что простой является задача кодирования такой модели.

Часто модель программной системы включает несколько классов, между которыми имеют место некоторые отношения. Двенадцатый раздел книги посвящён подробному изучению типов отношений, которые могут быть установлены между классами. В настоящем подразделе мы вкратце познакомимся только с одним из них — отношением типа *ассоциация*. Нам необходимы, по крайней мере, начальные сведения об отношении типа ассоциация для того, чтобы ввести классификацию полей, согласно которой поля разделяются на *поля, определяемые в классе*, и *поля, определяемые ассоциацией*.

Два класса находятся в отношении типа ассоциация в том случае, если *объекты этих классов объединяются (ассоциируются) в новую сущность*. Например, некоторые объекты класса `Man` (мужчина) могут быть ассоциированы с некоторыми объектами класса `Woman` (женщина), образуя новые сущности — семейные пары. В этом примере ранее независимые объекты классов `Man` и `Woman` образовали новые

сущности (семейные пары), состоящие из одного объекта класса `Man` и одного объекта класса `Woman`. Для моделирования такого характера отношений между классами и используется отношение типа ассоциация.

Рассмотрим ещё один пример. Пусть нашей задачей является моделирование структуры системы, которая включает: (1) сотрудников некоторой компании, (2) отделы этой же компании, а также (3) распределение сотрудников между отделами (описание того, какие сотрудники работают в каждом из отделов). Тогда модель системы может включать (1) класс `Employee` (наёмные работники), (2) класс `Department` (отделы) и (3) отношение ассоциации между классами `Employee` и `Department`, моделирующее распределение сотрудников между отделами. На рис. 11.19 приведена диаграмма классов, включающая классы `Employee` и `Department`, между которыми установлено отношение типа ассоциация.

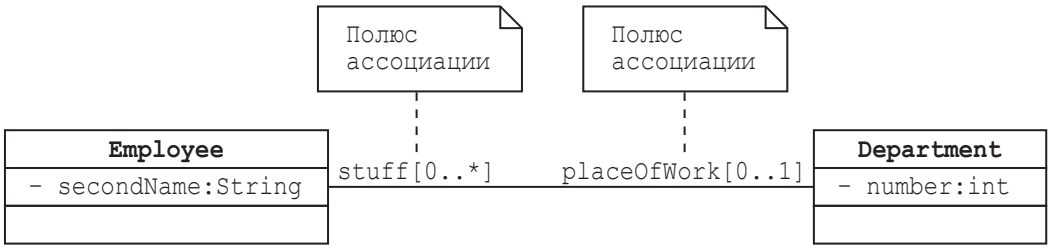



**Рис. 11.19.** Пример модели, включающей два класса, между которыми установлено отношение ассоциации

Графическим символом отношения ассоциации является отрезок прямой линии, соединяющий классы, объекты которых объединяются в новые сущности. Концы графического символа ассоциации, примыкающие к классам, называются *полюсами ассоциации*. Полюс ассоциации описывается именем и множественностью.

*Имя полюса* называет роль объектов класса, к которому примыкает полюс в новой сущности, образуемой в результате ассоциации. На рис. 11.19 полюс, примыкающий к классу `Employee`, назван `stuff` (персонал), а полюс, примыкающий к классу `Department`, назван `placeOfWork` (место работы).

*Множественность полюса* описывается такими же выражениями, которыми мы ранее описывали множественность полей, и специфицирует количество объектов класса, к которому примыкает полюс в новой сущности, образуемой в результате ассоциации. На рис. 11.19 полюс, примыкающий к классу `Employee`, имеет множественность, записанную в виде выражения `[0..*]`. Это означает, что в новой сущности, образуемой ассоциацией, может находиться ноль или некоторое количество объектов класса `Employee`. Иначе говоря, в любом отделе или не работает ни один наёмный работник из класса `Employee`, или работает некоторое количество наёмных работников этого класса. Аналогичным образом интерпретируется выражение, использованное для специфицирования множественности полюса, примыкающего к

классу Department. Множественность [0..1] означает, что в новой сущности, образуемой ассоциацией, может находиться ноль или один объект класса Department. Иными словами, любой наёмный работник из класса Employee может или работать в одном из отделов класса Department, или не работать в этом отделе, а одновременная работа в нескольких отделах запрещена.

В случае, когда ассоциируются только два класса, множественность полюса можно определить с использованием понятия «связь». Множественность полюса специфицирует *количество объектов класса, примыкающего к данному полюсу, связанных с одним объектом класса, примыкающего к противоположному полюсу*.

Имена stuff и placeOfWork, которые мы использовали для обозначения полюсов ассоциации на рис. 11.19, часто называются *полями, определяемыми ассоциацией*, поскольку они, по сути, имеют тот же смысл, что и *поля, определяемые в символе класса*.

При отображении полей, определяемых ассоциацией, в программный код учитываются их имена и множественность. Типом поля, определяемого ассоциацией, является имя ассоциированного класса. На рис. 11.20 приведен пример отображения диаграммы классов, приведенной на рис. 11.19, в код. Код, приведенный на рис. 11.20, не отражает явно новую сущность, образованную в результате ассоциации объектов классов Employee и Department. Эта сущность представлена неявно связями между объектами обоих классов, которые описаны в коде взаимными ссылками при помощи полей, определяемых ассоциацией.

```
class Employee {

   // описание полей
   private String secondName; // определено в символе класса
   private Department placeOfWork; // определено ассоциацией
            . . .
}

class Department {

   // описание полей
   private int number; // определено в символе класса
   private Employee[] stuff; // определено ассоциацией
            . . .
}
```

**Рис. 11.20.** Отображение полей, определяемых ассоциацией, в код

Наличие этих ссылок позволяет осуществлять навигацию в обоих направлениях: (1) от объектов класса Employee к объектам класса Department и (2) от объектов класса Department к объектам класса Employee.

### 11.3.5. Поля, определяемые рекурсивно

В ряде случаев возникает необходимость включать в список полей некоторого класса поля, имеющие тип, который определяется этим же классом. Такие поля называются полями, *определяемыми рекурсивно*. Рис. 11.21 иллюстрирует поля, определяемые рекурсивно.

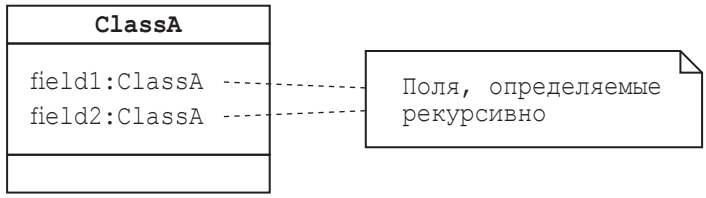



**Рис. 11.21.** Поля, определяемые рекурсивно

Необходимость использования рекурсивно определяемых полей возникает, как правило, в тех случаях, когда моделируемая сущность представляет собой некоторое количество связанных между собой объектов одного и того же класса. Рассмотрим класс, моделирующий дерево. Дерево представляет собой некоторое количество связанных между собой узлов. Все узлы дерева однотипны, а важным атрибутом любого узла является информация о его родительском узле. Поэтому, если мы включаем в список атрибутов класса, моделирующего дерево, поле, специфицирующее родительский узел, нам не обойтись без использования рекурсивно определяемого поля. Рис. 11.22 иллюстрирует использование рекурсивно определяемого поля для моделирования дерева.

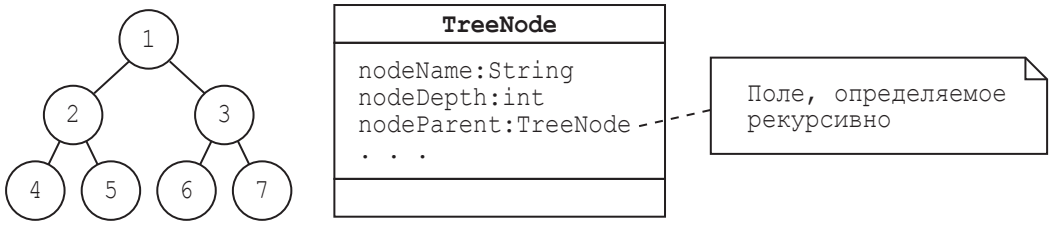



**Рис.11.22.** Пример использования рекурсивно определяемого поля при моделировании класса, объектами которого являются узлы дерева

В левой части рис. 11.22 изображен пример бинарного дерева, или дерева, для которого коэффициент ветвления равен двум. Это означает, что каждый родительский узел порождает точно два узла-потомка. Например, корневой узел 1 является родительским узлом для узлов-потомков 2 и 3. В правой части рис. 11.22 приведен графический символ класса `TreeNode` (узел дерева), моделирующий дерево. Модель представлена не полностью и содержит только некоторое количество полей атрибутивной модели.

Поле nodeName моделирует имя узла дерева. Поле nodeDepth моделирует глубину узла дерева или количество узлов от корня дерева до данного узла. Поле nodeParent представляет собой ссылку на родительский узел и определено рекурсивно.

### 11.3.6. Специфицирование начальных значений полей

Ранее, при изучении OCL-ограничений, мы познакомились с одним из способов специфицирования начального значения полей с помощью init-предложений. Так, например, начальное значение поля depth (глубина) класса Submarin (подводная лодка) может быть специфицировано при помощи следующего OCL-ограничения.

```
context Submarin::depth:float
init:   0.0
```

Начальные значения полей могут быть также специфицированы при описании поля непосредственно внутри графического символа класса. В этом случае начальное значение поля записывается в виде литерала в той же строке, в которой специфицируются имя и тип поля, справа от знака равенства. На рис. 11.23 приведена модель класса Submarin, в которой начальное значение поля depth специфицировано внутри графического символа класса.

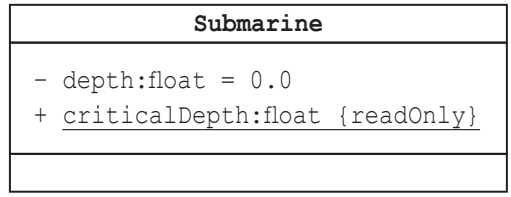
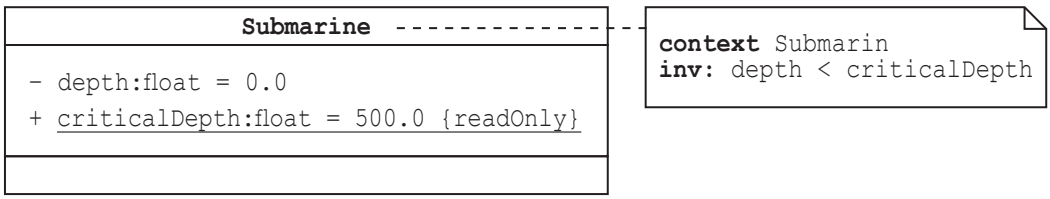



**Рис. 11.23.** Специфицирование начального значения поля в графическом символе класса

Если специфицируется начальное значение поля, помеченного ограничением {readOnly}, то это означает, что значение поля должно оставаться неизменным на протяжении всего времени существования любого объекта данного класса. Значение такого поля невозможно изменить, и оно, по сути, представляет собой поименованный литерал.

На рис. 11.24 приведена модель класса Submarin, которая ранее была изображена на рис. 11.18. В классе специфицированы начальные значения полей depth и criticalDepth. В нижней части рис. 11.24 приведен пример отображения этой модели в код.

Приведенный способ специфицирования начальных значений полей не является универсальным. Им удобно специфицировать поля с единичными значениями и примитивными типами: int, double, booleang и др.




```
class Submarine {

   // описание полей
   private float depth = 0.0;
   public static final float criticalDepth = 500.0;

   // описание методов
           . . .
}
```

**Рис. 11.24.** Специфицирование начального значения поля, помеченного ограничением {readOnly}

### 11.3.7. Специфицирование полей при помощи стереотипов

Стереотипы могут использоваться для уточнения полей и применяются тогда, когда важной является информация о принадлежности поля или группы полей к некоторому метаклассу полей. Например, при использовании класса для моделирования таблицы реляционной базы данных объект класса моделирует отдельную строку таблицы. В этом случае важной является информация о том, какие из полей класса формируют ключ для уникальной идентификации строки таблицы, или объекта класса. Принадлежность поля к метаклассу ключей может быть уточнена при помощи стереотипа. На рис. 11.25 приведена модель класса Person, рассматриваемого как таблица реляционной базы данных.

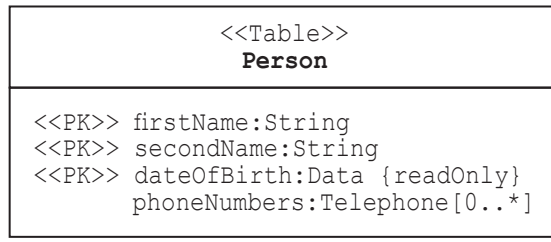



**Рис. 11.25.** Пример использования стереотипов для уточнения полей

При помощи стереотипа <<Table>> уточняется принадлежность класса Person к метаклассу Table (таблица), а при помощи стереотипа <<PK>> отмечены поля, которые выбраны в качестве *первичного ключа* (*Primary Key*), уникально идентифицирующего любой из объектов класса Person. Предполагается, что совокупное значение отмеченных трёх полей всегда уникально и поэтому может использоваться для идентификации объекта класса Person. Отметим также, что при моделировании таблицы реляционной базы данных используется только та часть класса, которую мы называем атрибутивной моделью. Таблица реляционной базы данных не обладает поведением и поэтому при её моделировании нет необходимости описывать методы.

Другим примером использования стереотипов для уточнения полей является случай, когда необходимо «навести порядок» в обширном списке полей. В этом случае поля можно сгруппировать и обозначить каждую группу при помощи стереотипа. На рис. 11.26 приведен пример атрибутивной модели класса Apartment (квартира), который может использоваться в системе учёта риэлторской компании.

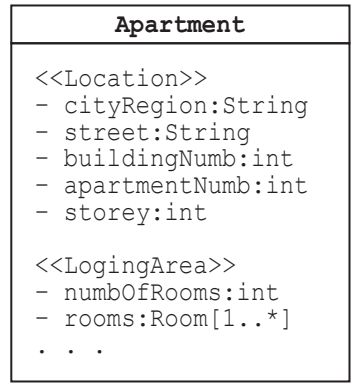



**Рис. 11.26.** Пример использования стереотипов для группировки полей

С целью группировки относительно большого количества полей класса Apartment использованы стереотипы <<Location>> (местонахождение) и <<LogingArea>> (жилые помещения). Стереотип <<Location>> группирует поля: cityRegion (район города), street (улица), buildingNumb (номер дома), apartmentNumb (номер квартиры) и storey (этаж), а стереотип <<LogingArea>> — поля: numbOfRooms (количество комнат), rooms (комнаты) и другие поля, обозначенные многоточием.

## 11.4. Описание методов

Отдельный метод моделирует способность объекта класса выполнять некоторую «работу» или реализовывать некоторое поведение, а совокупность методов

моделирует возможные поведения объектов класса. Каждый метод может иметь один или некоторое количество параметров.

Ранее мы познакомились со стандартными set- и get-методами, нестандартными методами, а также статическими и нестатическими методами. Более полный список методов, который нам понадобится для дальнейшего изложения, включает следующие виды методов:

- стандартные set- и get-методы и нестандартные методы;
- статические и нестатические методы;
- методы-конструкторы;
- перегруженные методы;
- абстрактные методы.

### 11.4.1. Стандартные и нестандартные методы

Мы неоднократно говорили о стандартных set- и get-методах. Резюмируем сказанное и рассмотрим, каким образом эти методы описываются в модели класса.

Во-первых, стандартные set- и get-методы обеспечивают внешний доступ к полям объекта. При помощи set-метода можно записать в поле новое значение, а при помощи get-метода — прочитать значение поля. Стандартные set- и get-методы необходимы для реализации идеи инкапсуляции, которая заключается в том, что непосредственный внешний доступ к полям должен быть запрещён и что получить этот доступ можно только опосредованно при помощи методов объекта. В этом смысле *стандартные set- и get-методы дополняют атрибутивную модель*, обеспечивая контролируемый доступ к полям извне. Ясно, что если внешний доступ к полям возможен только посредством стандартных set- и get-методов, то, исключив для некоторого поля set-метод мы лишаем внешнего пользователя объекта возможности безусловно изменять значение этого поля.

Во-вторых, стандартные set- и get-методы вызываются специальными сообщениями. Сообщение, при помощи которого вызывается set-метод, называется инструктивным сообщением, а сообщение, при помощи которого вызывается get-метод, — запросным сообщением.

В-третьих, имеется явная связь между наличием или отсутствием ограничения `{readOnly}`, которое мы применяем для описания полей, и использованием set- и get-методов. Если в описании поля отсутствует ограничение `{readOnly}`, то для обеспечения доступа к этому полю можно в список методов включить оба стандартных метода. Если же описание поля включает ограничение `{readOnly}`, то это поле предназначено только для чтения и в список методов включается только стандартный get-метод.

В-четвёртых, имеется связь между производными полями, OCL-ограничениями и использованием get-методов. Доступ к производным полям осуществляется только при помощи get-метода, который снабжается OCL-ограничением с body-предложением, в котором специфицируется способ формирования возвращаемого значения.

На рис. 11.27 приведен пример модели класса `Person`, которая включает описания стандартных set- и get-методов. Расстановка префиксов видимости в модели класса `Person` обеспечивает «почти» полную информационную скрытность. Исключение составляет поле `yearOfBirth`, снабженное `public` префиксом видимости. После начальной инициализации полей класса `Person` поле `yearOfBirth` превращается в поименованный литерал, доступ к которой может быть разрешен из любой точки программы.

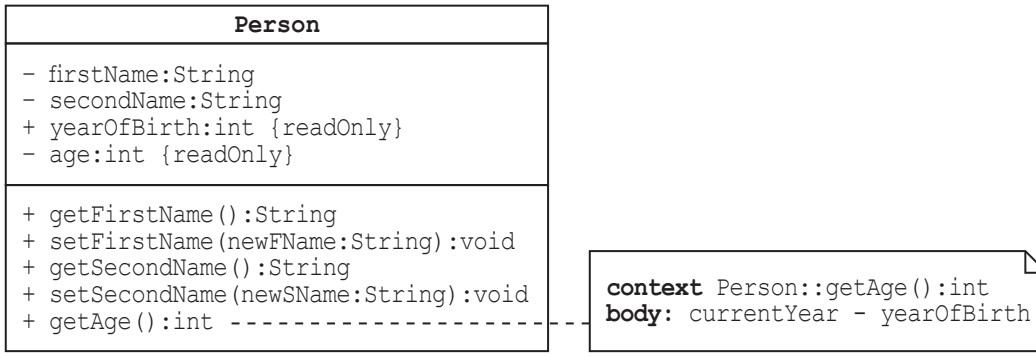



**Рис. 11.27.** Описание стандартных `set-` и `get-`методов

Модель иллюстрирует отмеченные особенности использования set- и get-методов. Анализируя модель, приведенную на рис. 11.27, легко заметить, что, несмотря на то, что оба поля `yearOfBirth` (год рождения) и `age` (возраст) снабжены `{readOnly}` ограничениями, поле `yearOfBirth` является базовым, а поле `age` — производным. Индикатором того, что поле `age` — производное, является OCL-ограничение, включённое в модель и специфицирующее способ формирования возвращаемого значения метода `getAge` при помощи body-предложения. Наличие `{readOnly}` ограничения в описании поля `age` означает, что оно предназначено для чтения и поэтому должно быть снабжено только get-методом.

Поля `firstName`, `secondName` и `phoneNumber` не снабжены `{readOnly}` ограничением. Это означает, что модель допускает изменение их значений в процессе «жизнедеятельности» объектов класса `Person`. Поэтому эти поля снабжены как set-, так и get-методами. Как видно на рис. 11.27, стандартные get-методы не имеют входных параметров, но обязательно имеют возвращаемое значение, а стандартные set-методы обязательно имеют входной параметр, но не обязательно снабжаются возвращаемым значением. На рис. 11.27 при описании всех set-методов использовано служебное слово `void`. Фактическое значение входного параметра set-метода представляет собой то новое значение, которое записывается в соответствующее поле.

Рис. 11.28 иллюстрирует возможное отображение модели, приведенной на рис. 11.27, в код.

Стандартные set- и get-методы, приведенные на рис. 11.28, обеспечивающие доступ к базовым полям, работают весьма примитивно. Get-методы *безусловно* возвращают значения соответствующих базовых полей, а set-методы *безусловно*

записывают значение своего параметра в соответствующее поле. Такое безусловное обеспечение доступа к базовым полям делает нерациональным использование стандартных методов, ибо исключает контроль над доступом к базовым полям со стороны класса. Контроль означает, что имеет место некоторое предусловие, которое должно проверяться перед вызовом стандартного метода. Если предусловие выполняется, то разрешается доступ к полю. В противном случае, система должна фиксировать исключительную ситуацию. Предусловия могут быть самыми разными. Например, объект может разрешать изменять значения своих базовых полей при помощи set-методов только в определённое время суток; объект может контролировать значения, которые записываются в его базовые поля при помощи set-методов и разрешать запись только определённых групп значений; объект может запрещать доступ к своим базовым полям со стороны некоторых «плохих» объектов.

```
class Person {

   // описание полей
   private String firstName;
   private String secondName;
   private final int yearOfBirth;
   private int age;

   // описание стандартных get- и set-методов
   public String getFirstName(); {
   return firstName;
   }
   public void setFirstName(String newFName); {
   firstName = newFName;
   }
   public String getSecondName(); {
   return secondName;
   }
   public void setSecondName(String newSName); {
   secondName = newSName;
   {
   public int getAge(); {
   currentYear = . . .;
   // вызов функции, возвращающей текущий год
   age = currentYear - yearOfBirth;
   return age;
   {
{
```

**Рис. 11.28.** Отображение модели, приведенной на рис. 11.27, в код

Отмеченные примеры являются примерами ограничений, накладываемых на стандартные get- и set-методы, и могут быть формально записаны в виде OCL-ограничений с пред- и постусловиями. Из проведенных рассуждений можно сделать вывод, что *в модель класса, содержащую стандартные get- и set-методы доступа к базовым полям, необходимо включить OCL-ограничения с pre- и post-условиями, специфицирующими условия доступа к этим полям*. Ниже приведен пример OCL-ограничения для метода `setFirstName`.

```
context Person::setFirstName(newFName:String):void
pre:    (currentTime > 23.30) and (currentTime < 6.30)
post:   getFirstName@pre():String < > getFirstName():String
```

В приведенном примере ограничению подвергается метод `setFirstName` класса `Person` (см. рис. 11.27). Предусловие разрешает выполнение этого метода только в ночное время, в промежуток времени с 23:30 и до 6:30. Постусловие проверяет, было ли фактически изменено значение поля `firstName`.

Стандартные get- и set-методы являются «продолжением» атрибутивной модели и необходимы только для обеспечения контролируемого доступа к полям. Они обеспечивают объектам класса стандартное поведение инкапсулированных объектов. Класс, в котором описаны только стандартные методы, не соответствует в полной мере ООП, поскольку объекты такого класса не обладают индивидуальным поведением.

*Индивидуальное поведение объекта определяется его нестандартными методами*. Нестандартные методы вызываются при помощи императивных сообщений, а правила их описания ничем не отличаются от правил описания стандартных методов, за исключением того, что имена нестандартных методов, как правило, не содержат префиксов get и set. Исключением являются get-методы-запросы.

Для более полного описания нестандартные методы должны быть уточнены при помощи OCL-ограничений с предусловиями и постусловиями.

В качестве примера рассмотрим модель класса `SimpleVoteMachine` (простая машина для голосования), приведенную на рис. 11.29.

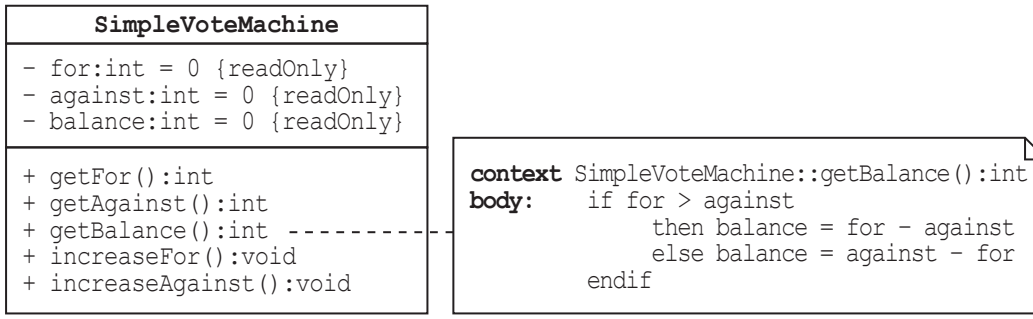



**Рис. 11.29.** Модель класса `SimpleVoteMachine`

Модель класса SimpleVoteMachine включает два базовых поля for (за) и against (против) и одно производное поле balance (баланс). Все три поля описаны ограничением {readOnly} и, следовательно, предназначены только для чтения. Поэтому эти поля снабжены только стандартными get-методами и не снабжены set-методами.

Способ формирования производного поля balance задаётся OCL-ограничением с body-предложением. Ограничение требует, чтобы общий баланс голосов (значение поля balance) формировался путем вычитания меньшего значения из большего значения для полей for и against.

Кроме стандартных get- и set-методов, класс SimpleVoteMachine содержит также два нестандартных метода increaseFor (увеличить «за») и increaseAgainst (увеличить «против»), определяющих индивидуальное поведение объектов этого класса. В приведенной модели индивидуальное поведение объекта класса SimpleVoteMachine заключается в умении регистрировать количество голосов «за» и «против» при каждом вызове методов increaseFor или increaseAgainst, соответственно.

```
class SimpleVoteMachine {
   private int for = 0;
   private int against = 0;
   private int balance = 0;
   // стандартные get-методы
   public int getFor(); {
      return for;
   }
   public int getAgainst(); {
      return against;
   }
   public int getBalance(); {
      if (for > against)
          balance = for - against;
      else
          balance = against - for;
      return balance;
   }
   // нестандартные методы
   public void increaseFor(); {
      for++;
   }
   public void increaseAgainst(); {
      against++;
   }
}
```

**Рис. 11.30.** Отображение модели, приведенной на рис. 11.29, в код

Предполагается, что машина для голосования снабжена сенсорным экраном, и каждое прикосновение к кнопке «за» на сенсорном экране машины приводит к вызову метода `increaseFor`, а каждое прикосновение к кнопке «против» — к вызову метода `increaseAgainst`.

Модель, приведенная на рис. 11.29, содержит достаточно информации для ручной или автоматической генерации кода. На рис. 11.30 приведен пример отображения модели класса `SimpleVoteMachine` в код.

### 11.4.2. Статические методы и методы-конструкторы

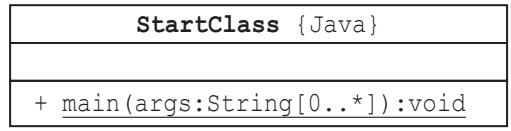
Статические методы определяют поведение класса как отдельной сущности. Статический метод «работает» со статическими полями и предназначен для реализации какой-либо функции класса, но не функции объекта этого класса. Описание метода как статического определяется разработчиком класса и зависит от его представления о роли метода в модели класса. Однако, существуют несколько особых случаев, когда метод класса *должен* быть объявлен как статический: (1) моделирование и кодирование `main`-метода класса; (2) моделирование и кодирование методов, предназначенных для работы с числовыми типами данных (например, метода, возвращающего квадратный корень числа, переданного ему в качестве параметра).

Работа любой объектной программы начинается с вызова метода, имеющего зарезервированное имя `main`. Чтобы активизировать объектную программу, необходимо указать имя какого-либо класса этой программы. При запуске программы система пытается обнаружить в указанном классе метод с именем `main` и передать ему управление. Поэтому хотя бы один класс объектной программы должен содержать `main`-метод. Когда начинает работать `main`-метод, то в памяти компьютера ещё нет ни одного объекта программы, и, следовательно, никакой другой метод не может быть вызван. По сути одна из задач `main`-метода — начать процесс создания объектов. Метод `main` имеет следующий стандартный заголовок, одинаковый для любого класса:

```
public static void main(String[] args)
```

Как видно из заголовка, метод `main` доступен из любой точки программы (служебное слово `public`), является статическим (служебное слово `static`), не возвращает значений (служебное слово `void`), имеет один входной параметр с именем `args`, который представляет собой одномерный массив данных типа `String`.

На рис. 11.31 приведен пример модели класса `StartClass` с `main`-методом, а также пример кода `main`-метода, работа которого заключается в выводе на экран монитора значений переданных ему параметров.




```
public static void main(String[] args){
   for (int i = 0; i < args.length; i++)
      System.out.print(args[i] + " ");
      System.out.println();
      }
```

**Рис. 11.31.** Статический метод `main` в модели класса и пример его кодирования

Язык программирования Java содержит предопределённые статические методы для реализации широкого спектра математических функций, таких, например, как нахождение значений тригонометрических функций, абсолютной величины числа, квадратного корня и др. Большинство этих методов находится в предопределённом классе с именем `Math`. На рис. 11.32 приведены заголовки некоторых статических методов класса `Math`.

```
class Math {
   . . .
   public static int abs(int a) {…}
   // возвращает абсолютное значение целочисленной переменной a

   public static double sqrt(double b) {…}
   // возвращает значение корня квадратного вещественной переменной b

   public static double sin(double c) {…}
   // возвращает значение синуса вещественной переменной c
```

**Рис. 11.32.** Примеры заголовков статических методов класса `Math`

В сообщении, при помощи которого вызывается статический метод, вместо ссылки на объект указывается имя класса. Структура сообщения для вызова статического метода имеет вид

```
<имя класса> . <имя статического метода>(<фактические параметры метода>)
```

Например, если переменная `sumOfSquares` хранит сумму квадратов катетов, то гипотенуза (значение переменной `hypotenuse`) может быть вычислена следующим образом:

```
double sumOfSquares = 24.31;
double hypotenuse = Math.sqrt(sumOfSquares);
```

Каждый класс может содержать один или несколько специальных методов, называемых *конструкторами*. Конструктор вызывается предложением со служебным словом `new`, при помощи которого создаётся новый объект класса. Конструкторы предназначены для начальной инициализации полей класса. Иными словами, конструкторы предназначены для задания начального состояния вновь созданного объекта. Конструкторы, так же, как и статические методы, характеризуют весь класс, в не его объекты. Однако при объявлении конструктора не используется служебное слово `static`. Заголовок конструктора немного отличается от заголовка обычного метода и формируется с учётом следующих правил: (1) имя конструктора всегда совпадает с именем класса; (2) в заголовке конструктора не указывается тип возвращаемого значения или служебное слово `void`.

*В UML-модели класса конструкторы явно не описываются*. Они появляются в программе при отображении модели в код. Однако модель класса может содержать комментарии и OCL-ограничения, специфицирующие способ инициализации полей, которые могут использоваться при кодировании конструктора. При начальной инициализации полей конструктор имеет наивысший приоритет среди всех способов инициализации. Например, если в коде при объявлении поля начальное значение задано при помощи литерала и это же поле инициализируется конструктором, то после завершения создания объекта в поле будет записано то значение, которое устанавливается конструктором. На рис 11.33 приведен код класса `Person`, модель которого изображена на рис. 11.27, снабжённый конструктором.

```
class Person {
   // описание полей
   private String firstName;
   private String secondName;
   private final int yearOfBirth;
   private int age;
   // описание конструктора
   Person(String initFirstName, String initSecondName, int initYearOfBirth)
   {
   firstName = initFirstName;
   secondName = initSecondName;
   yearOfBirth = initYearOfBirth;
   currentYear = // вызов функции, возвращающей текущий год
   age = currentYear – yearOfBirth;
   }
   // описание методов
       . . .
}
```

**Рис. 11.33.** Отображения класса `Person` в код с включением конструктора

Если в классе объявлен конструктор, то создание нового объекта этого класса осуществляется при помощи предложения со служебным словом `new`. Например:

```
Person boss = new Person("Сергей", "Дубинин", 1995);
```

В приведенном примере слева от символа «=» записывается тип и имя ссылки на новый объект, а справа — служебное слово `new`, затем имя конструктора и список фактических значений параметров конструктора (если используется конструктор с параметрами).

### 11.4.3. Перегруженные методы

Ранее мы, как правило, использовали термин «заголовок метода». Сейчас необходимо вспомнить понятие *сигнатура* метода, поскольку интерпретатор Java различает методы не по их заголовкам, а по сигнатуре. Сигнатурой метода называется часть заголовка метода, в которую входят *имя метода и список типов входных параметров*.

Например, заголовок метода-запроса `getArea`, который возвращает значение площади треугольника, получая в качестве входных параметров стороны треугольника (`sideA` и `sideB`), а также угол между ними (`angleAB`), может иметь вид:

```
public double getArea(float sideA, float sideB, Degree angleAB)
```

Сигнатурой метода `getArea` является следующая часть заголовка.

```
getArea(double, double, Degree)
```

ООП предполагает, что модель класса может включать несколько методов, имеющих одно и то же имя, но различную сигнатуру. Такие методы называются *перегруженными*. Перегруженные методы размещаются в одном и том же классе, имеют одинаковое имя, отличаются количеством и типами своих параметров и, самое главное, — кодом.

На рис. 11.34 приведен пример класса с именем `Item` (продаваемый товар), который использует два перегруженных метода: `discount` (скидка) и `totalItemsSold` (общее количество проданных товаров).

Метод `discount` позволяет уменьшить цену товара, определяемую значением поля `price`, и имеет две версии: (1) версия, в которой отсутствует входной параметр и стоимость товара уменьшается на регулярной основе, например, на 20% по истечении каждых трёх месяцев; (2) версия, использующая входной параметр `amount` (величина) и позволяющая уменьшить стоимость товара на произвольную величину, заданную в процентах.

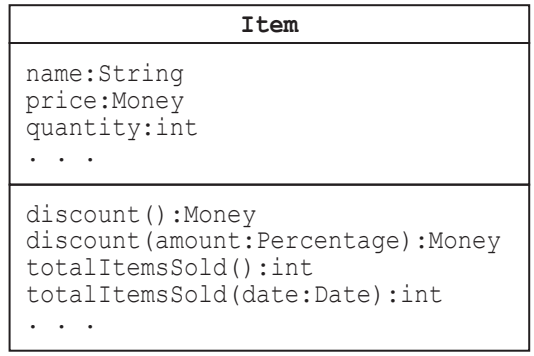



**Рис. 11.34.** Пример класса с двумя перегруженными методами `discount` и `totalItemsSold`

Метод `totalItemsSold` позволяет узнать общее количество проданных товаров и тоже имеет две версии: (1) версия, в которой используется входной параметр `date`, задающий дату, для которой определяется общее количество проданных товаров; (2) версия, не использующая входные параметры и определяющая общее количество проданных товаров для даты или срока, задаваемых «по умолчанию», например, в конце каждого месяца.

Предопределенные классы языка программирования Java часто включают группы перегруженных методов. Например, предопределенный класс `PrintStream`, моделирующий компьютерные консоли, содержит девять перегруженных методов с именем `print`:

```
public void print(boolean b)
public void print(char c)
public void print(int i)
public void print(long l)
public void print(float f)
public void print(double d)
public void print(char[] s)
public void print(String s)
public void print(Object obj)
```

Набор перегруженных методов `print` в классе `PrintStream` позволяет выводить на консоль данные любого из примитивных типов, а также массивов, строк и объектов классов, определяемых программистом, при помощи одного и того же сообщения `System.out.print(<список параметров>)`, вызывающего один из перегруженных методов `print`. При выполнении кода вызывается тот метод, у которого типы формальных параметров совпадают с типами фактических параметров, указанных в сообщении.

Перегрузку методов можно рассматривать как один из видов полиморфизма. Основные отличия перегруженных методов от полиморфных, так, как они

определены в подразделе 9.1.7, заключаются в следующем: (1) перегруженные методы располагаются в одном и том же классе, а полиморфные — в разных; (2) сигнатура перегруженных методов должна быть различной, а сигнатура полиморфных методов может быть одинаковой.

Часто, при кодировании класса в него вводят несколько *перегруженных конструкторов*. Это могут быть конструктор без параметров и несколько конструкторов с параметрами, отличающиеся количеством параметров. Перегруженные конструкторы позволяют формировать начальное состояние объекта в различных ситуациях.

Использование перегруженных методов позволяет разрабатывать более гибкие и универсальные модели классов.

### 11.4.4. Абстрактные методы и классы

В ряде случаев при разработке модели системы, состоящей из суперкласса и подклассов, *нецелесообразно* или даже *невозможно* детально разрабатывать один или несколько методов суперкласса. В этом случае методы описываются в модели (и кодируются в программе) только своим заголовком и носят наименование абстрактные.

Таким образом, *абстрактным методом* называется метод, который описан в коде программы только своим заголовком, а его тело отсутствует. Если класс включает хотя бы один абстрактный метод, то он называется *абстрактным классом*.

Ясно, что при помощи абстрактных классов нельзя создавать объекты, поскольку объект, созданный при помощи абстрактного класса, является «ущербным» в том смысле, что он не может выполнить сообщение, вызывающее абстрактный метод. Поэтому абстрактные классы не могут существовать самостоятельно. Абстрактные классы всегда имеют один или несколько подклассов, в которых реализуются его абстрактные методы. На рис. 11.35 приведена UML диаграмма классов, состоящая из четырёх классов и использованная ранее при изучении полиморфизма.

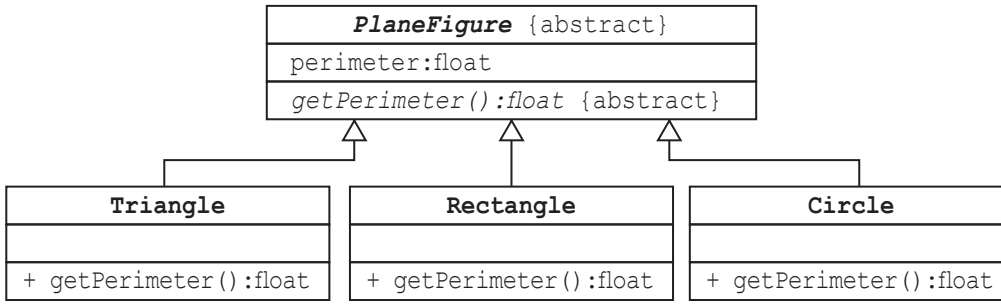



**Рис. 11.35.** Реализация абстрактного метода в подклассах

Диаграмма, приведенная на рис. 11.35, иллюстрирует способ описания абстрактных методов и классов на диаграмме классов. Как видно на рис. 11.35, на диаграмме классов имя абстрактного метода, так же, как и имя абстрактного класса, записывается курсивом и помечается ограничением {abstract}. В этом случае на диаграмме классов легко выделить абстрактные методы и классы. Однако, более важным является ограничение {abstract}, а использование курсива не обязательно.

Класс PlaneFigure (плоская фигура) на рис. 11.35 является абстрактным, поскольку содержит абстрактный метод getPerimeter (получить периметр). При помощи этого класса нельзя создавать объекты. Поэтому в систему включены несколько подклассов класса плоских фигур. Это подклассы Triangle (треугольник), Rectangle (прямоугольник) и Circle (окружность). В каждый из подклассов введен полиморфный метод getPerimeter, реализующий функции абстрактного метода суперкласса. Полиморфные методы имеют одинаковый заголовок, но различный код в каждом из подклассов. Код метода getPerimeter в подклассе пишется в соответствии с его специализацией. Метод getPerimeter в классе Triangle возвращает периметр треугольника, в классе Rectangle — периметр прямоугольника и т. д. Превратив класс PlaneFigure в абстрактный, мы создали модель *системы в которой существует только три класса плоских фигур*: треугольники, прямоугольники и окружности. Других плоских фигур в нашей модели нет. В модели системы плоских фигур на рис. 11.35 нельзя обойтись без абстрактного класса PlaneFigure. Эта модель не эквивалентна набору, состоящему из независимых классов Triangle, Rectangle и Circle, поскольку класс PlaneFigure, кроме абстрактного метода, содержит также не абстрактные члены, которые наследуются подклассами. Например, поле perimeter.

При отображении абстрактного метода и абстрактного класса в код используется служебное слово abstract, а вместо фигурных скобок, в которых размещаются предложения тела метода, записывается точка с запятой.

На рис. 11.36 приведен код абстрактного класса PlaneFigure.

```
abstract class PlaneFigure {

   // описание полей
   private float perimeter;

   // описание методов
   abstract float getPerimeter();
}
```

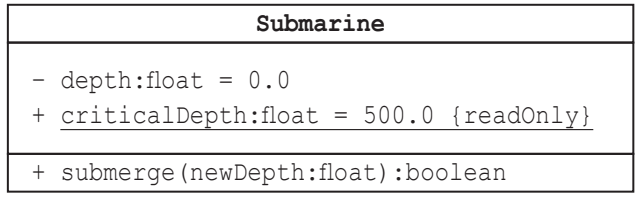
**Рис. 11.36.** Отображение абстрактного класса в код.

В языке программирования Java подклассы могут наследовать члены и реализовывать аабстрактные методы только одного суперкласса.

## 11.5. Моделирование исключительных ситуаций

Ранее было отмечено, что множество OCL-ограничений, специфицирующих инварианты класса, а также пред- и постусловия методов класса формируют контракт класса. Нормальная работа любого объекта класса должна осуществляться без нарушения контракта класса этого объекта. *Если контракт класса нарушается, то возникает исключительная ситуация*. Исключительная ситуация означает, что нарушена нормальная работа программной системы и необходимо вмешательство *обработчика исключительной ситуации*. В простейшем случае, при возникновении исключительной ситуации работа программы прерывается, а обработчик выводит на монитор сообщение, информирующее пользователя об исключительной ситуации. Например, для случая нарушения ограничения глубины погружения подводной лодки это сообщение может иметь вид: «глубина погружения превышает критическую».

На рис. 11.37 приведена модель класса `Submarine`, полученная из модели, приведенной на рис. 11.24 и снабжённая OCL-ограничениями, специфицирующими его контракт. Контракт представляет собой пред- и постусловия метода `submerge` (погружение). Логические выражения, использованные для записи пред- и постусловий метода `submerge`, могут быть сформулированы на русском языке следующим образом: (1) новая глубина, на которую требуется погрузить подводную лодку, должна быть меньше критической; (2) после погружения новая глубина должна быть больше предыдущей глубины.




Контракт класса Submarine

```
context Submarin::submerge(newDepth:float):boolean
pre:    newDepth < criticalDepth
post:   newDepth > depth
```

**Рис. 11.37.** Модель класса `Submarin`, включающая его контракт

Нарушение контракта класса `Submarine`, приведенного на рис. 11.37, возможно в двух случаях: (1) при нарушении предусловия и (2) при нарушении постусловия. При каждом нарушении предусловия возникает исключительная ситуация, принадлежащая некоторому классу исключительных ситуаций, который целесообразно назвать `IllegalDepthException` (исключение по недопустимой глубине), а

при каждом нарушении постусловия возникает исключительная ситуация, принадлежащая классу, который можно назвать NoSubmergeException (исключение по отсутствию погружения).

При отображении модели, приведенной на рис. 11.37, в программный код необходимо сделать следующее.

(1) В заголовке метода submerge перечислить имена классов исключительных ситуаций (IllegalDepthException и NoSubmergeException), которые могут возникнуть при нарушении контракта.
(2) В код метода submerge ввести предложения, осуществляющие создание нового объекта соответствующего класса исключительной ситуации каждый раз, когда возникает нарушение контракта.

На рис. 11.38 приведен код, иллюстрирующий средства, используемые в языке программировании Java для отслеживания исключительных ситуаций.

Как видно на рис. 11.38, классы исключительных ситуаций, которые могут возникнуть при работе метода, перечисляются в его заголовке после служебного throws (выбрасывать). Возникновение исключительной ситуации сопровождается созданием объекта соответствующего класса исключительных ситуаций при помощи предложения, начинающегося со служебного слова throw (выбросить).

```
class Submarine {

   // описание полей
   private float depth = 0.0;
   public static final float criticalDepth = 500.0;

   // описание метода submerge
   public boolean submerge(float newDepth)
                           throws IllegalDepthException,
NoSubmergeException {
      // проверка предусловия
      if (newDepth > criticalDepth) {
         throw new IllegalDepthException(newDepth);
      }
      depth = newDepth;

      // проверка постусловия
      if (newDepth < = depth) {
         throw new NoSubmeregeException(newDepth);
      }
   }
}
```

**Рис. 11.38.** Отображение контракта класса Submarin в код

В коде на рис. 11.38 создание объектов классов исключительных ситуаций осуществляется предложениями

```
throw new IllegalDepthException(newDepth);
throw new NoSubmeregeException(newDepth);
```

При создании объектов исключительных ситуаций используются конструкторы с параметрами, при помощи которых в объекты передается фактическое значение поля `newDepth`.

## 11.6. Моделирование интерфейсов

Интерфейс класса — это программная сущность, к определению которой можно подойти с нескольких сторон. Для подавляющего большинства случаев *интерфейс* класса можно определить как множество заголовков методов либо как абстрактный *класс, у которого все методы абстрактные*.

Интерфейс класса описывает поведение класса в терминах заголовков методов, отвечая на вопрос: «что умеет делать класс?» Однако, интерфейс не позволяет ответить на вопрос «Как класс реализует свои умения?» Поскольку методы интерфейса не имеют тела, то при помощи интерфейса, как и при помощи абстрактного класса, нельзя создавать объекты. Эта возможность предоставляется классам, которые *реализуют один или несколько интерфейсов*. В модель интерфейса нельзя включать никакие виды полей, а лишь статические литералы.

Интерфейс класса может рассматриваться как способ задания спецификаций, декларативно описывающих функции класса. Удобство в использовании интерфейсов заключается в том, что раздельно специфицируются функции класса и их конкретная реализация. Класс может реализовывать методы интерфейса любым способом, который, с точки зрения создателя класса, является наиболее предпочтительным. Поэтому метод, описанный в интерфейсе, может иметь различный код в различных классах.

Понятие интерфейс связано с понятием тип. Базовой структурной единицей объектной программы является класс, а одним из основных средств описания полей и данных является тип. Типы могут задаваться при помощи классов. Например, объявление класса с именем `BankAccount` (банковский счёт) позволяет использовать это имя в качестве имени типа. Однако, в ряде случаев полезно иметь возможность задавать типы более простым способом, без объявления класса. Интерфейс позволяет задать тип в абстрактной форме — в виде описаний заголовков методов и поименованных литералов.

Графический символ интерфейса тождественен графическому символу класса — это прямоугольник, разделенный на отделения горизонтальными линиями. Для подчёркивания того факта, что речь идёт не о классе, а об интерфейсе, графический

символ интерфейса снабжается стереотипом <<Interface>>. На рис. 11.39 приведен пример графического символа интерфейса.

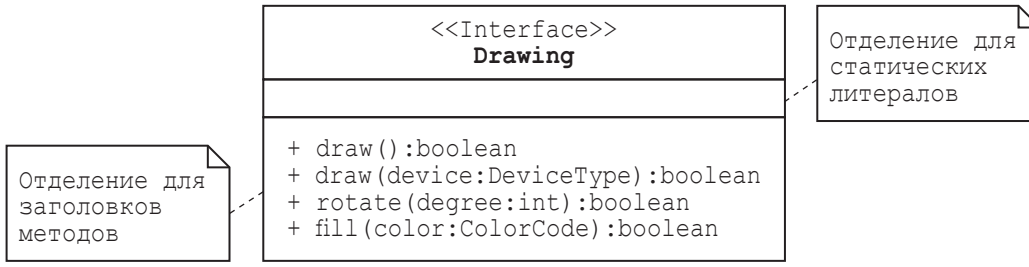



**Рис. 11.39.** Пример графического символа интерфейса

Интерфейс с именем Drawing (рисование), приведенный на рис. 11.39, декларативно специфицирует функции, обеспечивающие рисование и манипулирование рисунком, которые в дальнейшем могут быть реализованы в каком-либо из классов. Интерфейс представлен в виде набора заголовков методов. Перегруженный метод draw (рисовать) предполагает вывод изображения на периферийное устройство в двух случаях: (1) по умолчанию (например, на монитор); (2) на устройство, задаваемое входным аргументом device (оборудование). Метод rotate (повернуть) поворачивает изображение на угол degree (градус), а метод fill (заполнить) заполняет изображение цветом color (цвет). Как видно на рис. 11.39, все описанные в интерфейсе методы имеют префикс видимости public.

Отображение графического символа интерфейса в код включает заголовок интерфейса и тело интерфейса в фигурных скобках. В заголовке интерфейса записывается служебное слово interface, а тело состоит из перечня заголовков методов. Тела методов отсутствуют, а вместо тела метода записывается разделитель в виде символа «точка с запятой». На рис. 11.40 приведен код интерфейса, модель которого изображена на рис. 11.39.

```
interface Drawing {
   public boolean draw();
   public boolean draw(DeviceType device);
   public boolean rotate(int degree);
   public boolean fill(ColorCode color);
}
```

**Рис. 11.40.** Код модели интерфейса, изображенной на рис. 11.39

В том случае, когда необходимо обозначить, что некоторый класс реализует один или несколько интерфейсов, а подробное описание каждого из интерфейсов не обязательно, то используется нотация, приведенная на рис. 11.41.

Диаграмма, приведенная на рис. 11.41, моделирует класс с именем `DialogueAgent` (диалоговый агент), который реализует два интерфейса с именами `Sound` (звук) и `Text` (текст). Как видно на рис. 11.41, графический символ интерфейса может представлять собой небольшую окружность, которая соединяется с символом класса при помощи отрезка прямой.

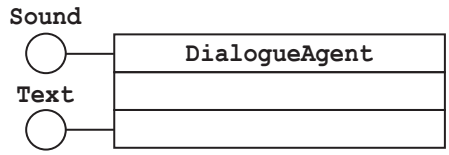



**Рис. 11.41.** Пример альтернативного способа изображения интерфейса

Отображение символа класса в программный код в том случае, когда класс реализует один или несколько интерфейсов, осуществляется следующим образом. В заголовок класса при помощи служебного слова `implements` (реализует) включается список реализуемых интерфейсов, а в теле класса кодируются все методы, описанные в интерфейсах. Так, например, диаграмма, приведенная на рис. 11.41, отображается в код так, как показано ниже.

```
class DialogueAgent implements Sound,Text {
// члены класса, которые обязательно включают коды всех
// методов интерфейсов Sound и Text
}
```

## 11.7. Вложенные классы

До сих пор мы относили к членам класса только поля и методы, однако ООП предполагает, что членами класса могут быть и другие классы. Если класс описан внутри другого класса и является его членом, то он называется *вложенным классом*. Использование вложенных классов обладает следующими достоинствами. Объекты вложенного класса находятся внутри инкапсулированного пространства внешнего класса. Поэтому методы объектов вложенного класса могут непосредственно обращаться как к членам вложенного класса, так и к остальным членам внешнего класса. На рис. 11.42 приведен пример изображения вложенности классов на диаграмме классов.

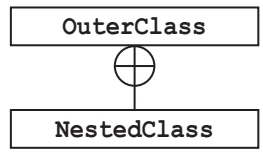



**Рис. 11.42.** Пример изображения вложенного класса

Класс с именем OuterClass является внешним, а класс с именем NestedClass — вложенным. Графический символ, представляющий собой окружность, описанную вокруг крестика, называется «якорь». Якорь указывает на внешний класс.

## 11.8. Наборы объектов в OCL

Поле с множественными значениями, а также полюс ассоциаций, в общем случае, специфицируют не один объект, а *набор объектов*. Поэтому для записи OCL-выражений, которые включают поля с множественными значениями и поля, определяемые ассоциацией, необходимо знать, каким образом в OCL представляются наборы объектов и какие операции предопределены для этих наборов.

При построении некоторых моделей возникает необходимость специфицировать запросы к системе с целью выбора набора объектов, удовлетворяющих заданным ограничениям. Например, может потребоваться выбрать из класса Person все объекты, для которых значение поля yearOfBirth (год рождения) равно или превышает 2000.

Для того чтобы можно было включать в OCL-ограничения выражения, оперирующие с наборами объектов, в состав языка объектных ограничений OCL включено несколько предопределенных типов наборов объектов и большое количество операций с наборами объектов.

### 11.8.1. Типы наборов объектов в OCL

На рис 11.43 приведена диаграмма классов, моделирующая структуру системы предопределенных типов наборов объектов, используемых в языке объектных ограничений OCL.

Как видно на рис. 11.43, структура типов наборов объектов в OCL представляет собой четыре реальных класса: Set (множество), OrderedSet (упорядоченное множество), Bag (мешок) и Sequence (последовательность), являющихся подклассами абстрактного суперкласса Collection (набор).

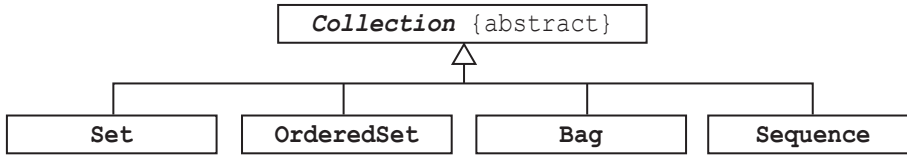



**Рис. 11.43.** Типы предопределенных наборов объектов в OCL

Абстрактный тип Collection используется для определения операций, общих для любого набора объектов, а в OCL-выражениях используются только типы реальных классов Set, OrderedSet, Bag и Sequence.

Набор объектов типа `Set` состоит из однотипных объектов и не содержит дубликатов объектов. Отдельный объект может входить в набор только один раз. Объекты в наборе типа `Set` располагаются в произвольном порядке и неупорядочены.

Набор объектов типа `OrderedSet` отличается от набора объектов типа `Set` только тем, что объекты в нем упорядочены. Упорядоченность объектов в этом наборе не следует понимать в смысле упорядоченности, полученной в результате сортировки (размещение объектов в порядке возрастания либо убывания их значений). Упорядоченность объектов в наборе `OrderedSet` следует понимать в смысле очередности. Для всех объектов набора `OrderedSet` известно, какой объект предшествует данному объекту, а также какой объект следует за данным объектом.

Набор объектов типа `Bag` отличается от набора типа `Set` тем, что в нем допустимы копии объектов. Один и тот же объект может входить в набор типа `Bag` несколько раз. Объекты в наборе типа `Bag` располагаются в произвольном порядке и неупорядочены.

Набор объектов типа `Sequence` отличается от набора типа `Bag` тем, что объекты в нем упорядочены. Упорядоченность объектов в этом наборе следует понимать так же, как и упорядоченность объектов в наборе типа `OrderedSet`.

В том случае, когда набор состоит из объектов примитивных типов (числа, булевы данные и символы), а также из строк символов, то набор объектов может быть задан перечислением значений объектов в фигурных скобках. Имя типа набора помещается перед фигурными скобками. Например:

```
Set{1, 2, 88, 7}
Bag{1, 2, 88, 2}
Set{'red', 'green', 'black', 'yellow'}
OrderedSet{'red', 'orange', 'yellow', 'green'}
Sequence{1.21, 3.44, 9.81, 7.42}
```

В том случае, когда перечислением задается набор, состоящий из непрерывной последовательности целых чисел, допускается не перечислять все числа, а указать первое и последнее числа последовательности, разделенные двоеточием. Например, набор

```
Set{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

можно записать в виде

```
Set{1..10}
```

Задание набора объектов путем перечисления их значений используется, главным образом, в OCL-ограничениях с init-предложением для задания начальных значений полей и полюсов ассоциаций со множественными значениями. Например,

список авторов, моделируемый полем `authors:String[1..*]`, может быть задан следующим OCL-ограничением:

```
context Book::authors:String[1..*]
init:   Set{'Ильф', 'Петров'}
```

В OCL-выражениях, которые используются во всех остальных видах OCL-ограничений, наборы задаются *неявно в виде имени поля или полюса ассоциации со множественными значениями*.

### 11.8.2. Базовые операции с наборами объектов в OCL

Все операции OCL, допустимые для наборов объектов, обладают следующим фундаментальным свойством. *Операция не изменяет набор*, к которому она применяется. Исходный набор объектов всегда остается неизменным, а в некоторых случаях создается новый набор. Современная версия OCL включает несколько десятков предопределенных операций над наборами объектов. Некоторые операции применимы для всех типов наборов, а некоторые — специализированы для конкретных типов. В таблице на рис. 11.44 приведен перечень операций, применимых для всех типов наборов объектов.

Операция «равенство» определяет равенство различным образом, в зависимости от типа набора.

Два набора объектов типа `Set` считаются равными, если все объекты первого набора присутствуют во втором наборе и наоборот.

Два набора объектов типа `OrderedSet` считаются равными, если (1) они равны в смысле наборов типа `Set` и (2) порядок размещения объектов в обоих наборах также совпадает.

Два набора объектов типа `Bag` считаются равными, если (1) все объекты первого набора присутствуют во втором наборе и наоборот, а также (2) количество одинаковых объектов в обоих наборах совпадает.

Два набора объектов типа `Sequence` считаются равными, если они равны в смысле наборов типа `Bag` и порядок размещения объектов в обоих наборах также совпадает.

Операция «неравенство» противоположна операции «равенство».

Операция `including(object)` работает следующим образом.

Для наборов объектов типа `Bag` операция возвращает новый набор, полученный из исходного путем добавления в него объекта `object`.

Для наборов объектов типов `Set` и `OrderedSet` объект `object` добавляется в результатный набор только в том случае, если он отсутствует в исходном наборе. В противном случае операция возвращает исходный набор.

Для упорядоченных наборов объектов типов `OrderedSet` и `Sequence` новый объект `object` добавляется в конец набора.

| Имя операции в OCL-выражении | Описание операции |
|---|---|
| = | Операция «равенство». Сравнивает два набора объектов и возвращает значение true, если наборы равны. |
| <> | Операция «неравенство». Сравнивает два набора объектов и возвращает значение true, если наборы не равны. |
| size() | Определяет количество объектов в наборе |
| isEmpty() | Возвращает значение true, если набор объектов не содержит ни одного объекта. |
| notEmpty() | Возвращает значение true, если набор объектов содержит один или несколько объектов. |
| count(object) | Возвращает количество объектов object в наборе. |
| including(object) | Возвращает новый набор объектов, полученный из исходного набора путем добавления в него объекта object. |
| excluding(object) | Возвращает новый набор объектов, полученный из исходного набора путем исключения из него объекта object. |
| includes(object) | Возвращает значение true, если набор содержит объект object. |
| excludes(object) | Возвращает значение true, если набор не содержит объект object. |
| includesAll(collection) | Возвращает значение true, если набор содержит все объекты из collection. |
| excludesAll(collection) | Возвращает значение true, если набор не содержит ни одного объекта из collection. |
| sum() | Для наборов числовых типов возвращает сумму всех значений объектов. |

**Рис. 11.44.** Базовые предопределенные операции OCL над наборами объектов

Операция excluding(object) возвращает новый набор объектов, полученный из исходного набора путем удаления из него объекта object. Для наборов объектов Set и OrderedSet удаляется только один объект, а для упорядоченных наборов типа OrderedSet и Sequence удаляются все копии объекта object.

При записи операции над набором объектов в OCL-выражениях используется следующая конструкция:

```
<collection> -> <operation>(<parameter>)
```

collection — спецификация набора объектов;
operation — имя операции;
parameter — параметр операции.

Все операции над наборами объектов обозначаются при помощи символа стрелки, который часто составляется из двух символов «минус» и «больше». Стрелка направлена слева направо. Слева от символа стрелки *специфицируется набор объектов, к которому применяется операция в виде имени поля или полюса ассоциации со множественными значениями*. Справа от символа стрелки записывается имя операции с входным параметром в круглых скобках. В тех случаях, когда используется операция без параметров, скобки остаются пустыми.

Рассмотрим пример, иллюстрирующий использование операций `size` в OCL-выражении. На рис. 11.45 приведена диаграмма классов, моделирующая структуру приложения, предназначенного для автоматизированного учета продажи билетов на авиарейсы.

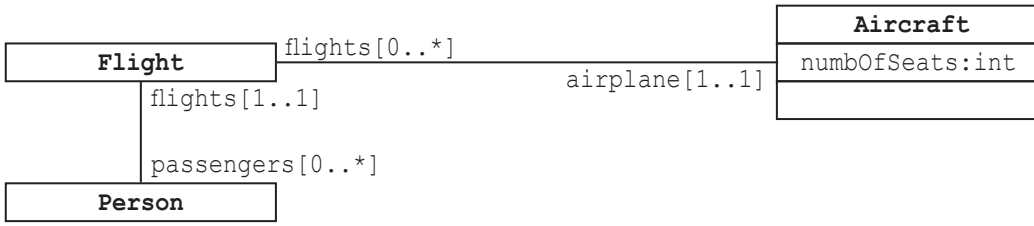



**Рис. 11.45.** Система учета продажи билетов на авиарейсы

Система состоит из трех ассоциированных классов: `Flight` (рейс), `Aircraft` (летательный аппарат) и `Person` (личность).

Класс `Flight` моделирует множество авиарейсов. Наибольшее количество билетов, которое может быть продано на авиарейс, определяется типом летательного аппарата, закрепленного за этим рейсом.

Множество доступных летательных аппаратов моделируется классом `Aircraft`, ассоциированным с классом `Flight`. Полюса этой ассоциации интерпретируются следующим образом. Полюс `airplane[1..1]` означает, что с одним объектом класса `Flight` связан только один объект класса `Aircraft` (рейс выполняется одним летательным аппаратом), а полюс `flights[0..*]` означает, что с одним объектом класса `Aircraft` может быть связано несколько объектов класса `Flight` (один и тот же летательный аппарат может использоваться при выполнении нескольких различных рейсов).

Пассажиры, участвующие в авиарейсах, моделируются классом `Person`, который также ассоциирован с классом `Flight`. Полюс `flights[1..1]` означает, что с одним объектом класса `Person` связан только один объект класса `Flight` (в данном, конкретном рейсе принимает участие одна личность из класса `Person`), а полюс `passengers[0..*]` означает, что с одним объектом класса `Flight` может быть связано несколько объектов класса `Person` (в одном авиарейсе может принимать участие некоторое количество личностей).

При учете количества проданных билетов естественным является следующее ограничение: «*на авиарейс может быть продано такое количество билетов,*

*которое не превышает количества пассажирских сидений в летательном аппарате, закрепленном за данным рейсом*». Это ограничение должно быть истинным для любого объекта класса `Flight` и может быть специфицировано при помощи следующего инварианта:

```
context Flight
inv:    self.passengers -> size()<= airplane.numbOfSeats
-- numbOfSeats поле класса Aircraft
```

Булево выражение в приведенном инварианте представляет собой неравенство. Левая часть неравенства — это результат применения операции `size` к набору объектов типа `Person`, который специфицирован именем поля класса `Flight` (или именем полюса ассоциации) `passengers`. Операция `size` возвращает количество объектов класса `Person`, ассоциированных с одним объектом класса `Flight`. Правая часть неравенства — это, по сути, значение поля `numbOfSeats` (количество кресел) класса `Aircraft`. Поскольку поле `numbOfSeats` не принадлежит классу `Flight`, то для доступа к нему необходимо использовать составное имя, состоящее из имени объекта класса `Aircraft` (или имени полюса ассоциации `airplane`) и имени поля `numbOfSeats`.

### 11.8.3. Итерационные операции с наборами объектов в OCL

Часто операции с наборами объектов являются итерационными. Итерационная операция последовательно получает доступ ко всем объектам набора и оценивает их при помощи параметра операции. Параметр итерационной операции, в общем случае, представляет собой выражение, которое соответствует характеру операции. Например, для того, чтобы выбрать из класса `Person` все объекты, для которых значение поля `yearOfBirth` (год рождения) равно или превышает `2000`, понадобится итерационная операция, параметр которой представляет собой выражение, детерминирующее требуемые объекты. Современная версия OCL предопределяет десятки итерационных операций для наборов объектов. В таблице на рис. 11.46 приведен перечень основных итерационных операций, применимых для всех типов наборов объектов.

При записи итерационной операции с набором объектов в OCL-выражениях чаще всего используется следующая конструкция:

```
<collection> -> <operation>(<expr>)
```

`collection` — спецификация набора объектов;
`operation` — имя итерационной операции;
`expr` — выражение-параметр итерационной операции.

В параметр итерационной операции может быть введена *итерационная переменная*, специфицирующая объекты набора, которые оцениваются при помощи параметра. Тип итерационной переменной всегда совпадает с типом объектов набора, поэтому при записи OCL-выражений его не указывают. В случае использования итерационной переменной, итерационная операция специфицируется при помощи одной из следующих конструкций:

```
<collection> -> <operation>(<iteration variable>:<type>|<expr>)
<collection> -> <operation>(<iteration variable>|<expr>)
```

`iteration variable` — имя итерационной переменной;
`type` — тип итерационной переменной;
`expr` — выражение-параметр итерационного оператора.

| Имя итерационной операции | Описание операции |
|---|---|
| `any(expr)` | Возвращает случайно выбранный объект набора, для которого значение `expr` равно `true`. |
| `collect(expr)` | Возвращает новый набор, созданный из исходного набора в соответствии со значением `expr`. |
| `exists(expr)` | Возвращает значение `true`, если хотя бы для одного объекта набора значение `expr` равно `true`. |
| `forAll(expr)` | Возвращает значение `true`, если для всех объектов набора значение `expr` равно `true`. |
| `isUnique(expr)` | Возвращает значение `true`, если для всех объектов набора значение `expr` является уникальным. |
| `one(expr)` | Возвращает значение `true`, если только для одного объекта набора значение `expr` равно `true`. |
| `select(expr)` | Возвращает те объекты набора, для которых значение `expr` равно `true`. |
| `reject(expr)` | Возвращает те объекты набора, для которых значение `expr` равно `false`. |
| `sortedBy(expr)` | Возвращает объекты набора, отсортированные в соответствии со значением `expr`. |
| `iterate(. . .)` | Выполняет итеративную операцию над всеми объектами набора. |

**Рис. 11.46.** Предопределенные итерационные операции OCL над наборами объектов

В ряде случаев введение итерационной переменной в параметр является избыточной детализацией описания итерационной операции и усложняет чтение и OCL-ограничения и понимание его смысла.

**Операция `any`** позволяет выбрать один, произвольный объект из набора объектов, который удовлетворяет условию, сформулированному в виде выражения-параметра `expr`. Выражение-параметр операции `any` является булевым выражением, которое принимает значение `true` в том случае, когда оцениваемый объект набора удовлетворяет условию выбора. Если ни один из объектов набора не удовлетворяет условию выбора, то результат операции `any` является неопределенным.

Рассмотрим применимость операции `any` на примере. Уточним модель учета продажи билетов на авиарейсы, приведенную на рис. 11.45, следующим образом. Введем в класс `Flight` новый метод-запрос, возвращающий ссылку на пассажира, которому авиакомпания вручает приз. Будем считать, что авиакомпания вручает приз только одному из пассажиров авиарейса, случайно выбираемому среди тех, кто в течение года пролетел более 9999 км. Требуемое уточнение модели можно осуществить при помощи следующего OCL-ограничения с def-предложением.

```
context Flight
def:getWinner():Person =
self.passengers -> any(passengers.yearDistance > 9999)
-- yearDistance – поле класса Person
```

Ограничение вводит в класс `Flight` новый метод-запрос с именем `getWinner` (получить победителя), который осуществляет выбор пассажира в соответствии с требуемым условием. При описании способа формирования возвращаемого значения метода `getWinner` использована итерационная операция `any`, которая применена к набору объектов `passengers`. Этот набор представляет собой те объекты класса `Person`, которые ассоциированы с одним объектом класса `Flight`. По сути, набор объектов `passengers` моделирует пассажиров авиарейса. Параметр-выражение операции `any` принимает значение `true` во всех случаях, когда значение поля `yearDistance` (годовое расстояние) для объекта класса `Person` больше 9999. Составное имя поля `passengers.yearDistance` определяет навигацию из класса `Flight`, указанного в контексте, к классу `Person`. Вопросы навигации между ассоциированными классами будут изучаться в последующем разделе.

**Операция `collect`** позволяет сформировать новый набор объектов из исходного набора объектов. Отличительной особенностью операции `collect` является то, что *тип нового набора объектов отличен от типа исходного набора объектов*. Иными словами новый набор не является поднабором исходного набора. Объекты нового набора формируются из объектов исходного набора в соответствии со значением парамерта операции `collect`.

Рассмотрим пример использования операции `collect`. Уточним модель учета продажи билетов на авиарейсы, приведенную на рис. 11.45, следующим образом. Введем в класс `Flight` метод-запрос, формирующий и возвращающий набор,

состоящий из фамилий всех пассажиров, участвующих в рейсе. Требуемое уточнение модели можно сделать при помощи следующего OCL-ограничения:

```
context Flight
def:getNames():Bag(String) =
self.passengers -> collect(passengers.name)
-- name – поле класса Person
```

Ограничение вводит в класс `Flight` метод-запрос с именем `getNames` (получить фамилии), который возвращает набор объектов типа `String`. Тип всего набора `Bag`, поскольку пассажиры могут иметь совпадающие имена. При описании способа формирования возвращаемого значения метода `getNames` использована итерационная операция `collect`, которая применена к набору объектов `passengers`, моделирующего пассажиров конкретного рейса. Параметром оператора `collect` является поле `name` класса `Person`. Из каждого объекта типа `Person` операция `collect` формирует объект типа `String`, значением которого является фамилия пассажира. Имя поля `passengers.name` является составным и определяет навигацию из класса `Flight`, указанного в контексте, к классу `Person`.

**Операция `exists`** позволяет определить, присутствует ли в наборе хотя бы один объект, удовлетворяющий условию, сформулированному при помощи параметра `expr`. Операция `exists` возвращает значение `true`, если искомый объект присутствует в наборе, и значение `false`, если искомый объект отсутствует в наборе.

Рассмотрим пример использования операции `exists`. Введем в класс `Flight` на рис. 11.45 еще один метод-запрос, позволяющий ответить на вопрос: «Участвует ли в рейсе пассажир по фамилии Жванецкий». OCL-ограничение, специфицирующее требуемый метод, имеет вид:

```
context Flight
def:getPassenger(who:String):boolean =
self.passengers -> exists(passengers.name = 'Жванецкий')
-- name – поле класса Person
```

Ограничение вводит в класс `Flight` метод-запрос с именем `getPassenger` (получить пассажира), с входным параметром типа `String`, который возвращает значение булевого типа. При описании способа формирования возвращаемого значения в методе `getPassenger` использована итерационная операция `exist`, которая применена к набору объектов `passengers`, моделирующего пассажиров конкретного рейса. Параметр-выражение операции `exist` принимает значение `true` во всех случаях, когда значение поля `name` для объекта класса `Person` равно литералу `'Жванецкий'`

**Операция `forAll`** позволяет определить, удовлетворяют ли все объекты набора условию, сформулированному при помощи параметра операции `expr`. Оператор `forAll` возвращает значение `true` только в том случае, когда все объекты набора удовлетворяют условию. Если хотя бы один из объектов не удовлетворяет условию, то операция `forAll` возвращает значение `false`.

Рассмотрим следующее OCL-ограничение, уточняющее модель, приведенную на рис. 11.45, и иллюстрирующее использование операции forAll.

```
context Flight
inv: self.passengers ->
forAll(passengers.registration = 'зарегестрирован')
-- registration – поле класса Person
```

Приведенное ограничение специфицирует инвариант класса Flight, который на русском языке может быть сформулирован следующим образом: «Все пассажиры рейса должны быть зарегистрированы». В OCL-выражении инварианта операция forAll применена к набору объектов passengers. Параметр операции представляет собой булево выражение, возвращающее значение true, если значение поля registration класса Person равно 'зарегестрирован'.

Операции forAll и exists логически взаимозаменяемы и могут использоваться для записи OCL-выражений, имеющих одинаковый смысл. Приведенные ниже конструкции логически эквивалентны.

```
<collection> -> exists(<expr>)
not <collection> -> forAll(not <expr>)
```

**Операция isUnique** позволяет ответить на вопрос: «Обладают ли все объекты набора уникальным признаком?» Для каждого объекта набора операция isUnique проверяет, уникальна ли его характеристика, заданная параметром операции, и возвращает значение true, если все объекты набора имеют различные значения отмеченной характеристики. Если среди объектов набора присутствуют хотя бы два объекта с совпадающими значениями характеристики, операция isUnique возвращает значение false.

Ранее, в подразделе 9.1.3, мы отметили, что одним из способов реализации объектной идентичности является использование ключа класса. Ключом класса мы называем те поля класса, совокупное значение которых уникально идентифицирует объект. Операция isUnique может быть использована для проверки того, являются ли выбранные поля класса его ключом. Проиллюстрируем применимость оператора на примере уточнения модели, приведенной на рис. 11.45. Введем в класс Flight метод-запрос, который проверяет, являются ли уникальными имена и фамилии пассажиров авиарейса. Иными словами, новый метод-запрос должен ответить на вопрос о том, могут ли поля класса Person, хранящие имя и фамилию, быть ключом для набора объектов, моделирующих пассажиров авиарейса. Специфицируем этот метод при помощи следующего OCL-ограничения:

```
context Flight
def:getUniqueness():boolean =
self.passengers -> isUnique(passengers.fName.concat(passengers.sName))
-- firstName и secondName- поля класса Person
```

Приведенное ограничение вводит в класс Flight новый метод-запрос с именем getUniqueness (получить уникальность), который осуществляет проверку уникальности имени и фамилии пассажиров авиарейса. При описании способа формирования возвращаемого значения метода getUniqueness использована итерационная операция isUnique, которая оперирует с объектами набора passengers. Параметр операции isUnique предсталяет собой конкатенацию полей firstName (имя) и secondName (фамилия) класса Person. Поскольку класс Person не указан в контексте, использованы составные имена для полей firstName и secondName.

**Операция one** позволяет ответить на вопрос: «Есть ли среди объектов набора в точности один, удовлетворящий заданному условию?» Условие задается булевым выражением-параметром оператора one. Операция one проверяет каждый из объектов набора на соответствие заданному условию и возвращает значение true, если соответствие имеет место в точности для одного объекта. Если среди объектов набора нет ни одного объекта, соответствующего заданному условию, или имеется несколько таких объектов, то операция one возвращает значение false.

**Операция select** формирует новый набор объектов из исходного набора. Новый набор является частью исходного набора и имееет тип исходного набора. Параметром операции select является булево выражение, которое специфицирует критерий выбора объектов их исходного набора. Вновь сформированный набор содержит только те объекты из исходного набора, для которых значение параметра оператора select равно true.

Проиллюстрируем применимость операции select на примере системы учета продажи билетов на авиарейсы (см. рис. 11.45). Введем в класс Flight метод-запрос, возвращающий ссылки на пассажиров, которые, воспользовавшись услугами авиакомпании, в течение года пролетели более 9 999 км. Такое уточнение модели можно сделать при помощи следующего OCL-ограничения:

```
context Flight
def:getWinners():Set(Person) =
self.passengers -> select(passengers.yearDistance > 9999)
-- yearDistance – поле класса Person
```

Ограничение вводит в класс Flight новый метод-запрос с именем getWinners (получить победителей), который возвращает набор ссылок на объекты типа Person. Предполагается, что в этом наборе нет одинаковых объектов, поэтому набор возвращаемых ссылок имеет тип Set. При описании способа формирования возвращаемого значения метода getWinners использована итерационная операция select, которая оперирует с каждым объектом набора passengers. Напомним, что набор объектов passengers моделирует пассажиров авиарейса. Параметр-выражение операции select принимает значение true во всех случаях, когда значение поля yearDistance (годовое расстояние) для объекта класса Person больше 9999.

**Операция reject** является, в некотором смысле, противоположной операции select. Она позволяет сформировать новый набор объектов из исходного набора,

который является частью исходного набора и объекты которого имеют тип исходного набора. Параметром операции `reject` является булево выражение, которое специфицирует критерий выбора объектов из исходного набора. Вновь сформированный набор содержит те объекты из исходного набора, для которых значение параметра операции `reject` равно `false`.

**Операция `iterate`** является наиболее фундаментальной и наиболее общей итерационной операцией. Все ранее изученные итерационные операции могут рассматриваться как частные случаи операции `iterate`. Операция `iterate` имеет следующую структуру:

```
<collection> -> iterate(<variable>:<vType>;
                        <result>:<rType> = <initialValue>|
                        <expr-with-variable-and-result>)
```

`collection` — спецификация набора объектов;
`variable` — имя итерационной переменной;
`vType` — тип итерационной переменной;
`result` — переменная-аккумулятор результата итерационной операции;
`rType` — тип переменной-аккумулятора;
`initialValue` — выражение, задающее начальное значение аккумулятора;
`expr-with-variable-and-result` — выражение-параметр итерационной операции.

Перед началом выполнения операции `iterate` в переменную-аккумулятор помещается начальное значение. Выражение-параметр итерационной операции специфицирует действия, которые при каждой итерации выполняются над содержимым аккумулятора и значением итерационной переменной. На каждой итерации итерационная переменная ссылается на очередной объект набора. Результат выполнения действия, задаваемого выражением-параметром, записывается в аккумулятор, заменяя в нем предыдущее значение.

Проиллюстрируем применимость операции на примере OCL-выражения, специфицирующего итерационный процесс нахождения суммы множества целых чисел:

```
Set{1-1000} -> iterate(i:int; sum:int = 0 | sum +i)
```

В приведенном примере набор объектов представляет собой множество целых чисел от 1 до 100. Целочисленная итерационная переменная с именем `i` на каждой итерации принимает значение очередного целого числа. Переменная-аккумулятор с именем `sum` принимает начальное значение, равное 0. Выражение-параметр задает действие, заключающееся в суммировании значения аккумулятора со значением итерационной переменной.

Рассмотрим еще один пример. Введем в систему учета продажи билетов на авиарейсы (см. рис. 11.45) метод-запрос, возвращающий ссылки на всех

пассажиров-женщин, участвующих в авиарейсе. OCL-ограничение, специфицирующее этот метод-запрос, приведено на рис. 11.47.

```
context Flight
def:getWomen():Set(Person) =
self.passengers -> iterate(prs:Person; resultSet:Set(Person) = Set{}|
                           if passengers.gender = 'female'
                              then resultSet.including(prs)
                              else resultSet
                           endif
-- gender – поле класса Person
```

**Рис. 11.47.** Пример использования операции `iterate` для специфицирования метода-запроса класса `Flight` (см. рис. 11.45)

Метод `getWomen` (получить женщин) возвращает множество ссылок на объекты класса `Person`. Ссылки формируются оперцией `iterate`. Операция осуществляет итерационную обработку объектов набора `passengers`, моделирующего пассажиров авиарейса. Итерационная переменная этой операции с именем `prs`, является ссылочной переменной типа `Person`. Переменная-аккумулятор с именем `resultSet` (результатный набор) представляет собой `Set`-набор объектов типа `Person`. Начальное значение аккумулятора — пустое множество, описано в виде `Set{}`. Выражение-параметр специфицировано при помощи «if-then-else» операции, которая добавляет новую ссылку в аккумулятор (`resultSet.including(prs)`) только в том случае, если значение поля `gender` (пол) принимает значение `'female'`.

## Упражнения для практических занятий

11.1. Разработайте модель класса `Patient` (пациенты), состоящую только из атрибутивной части. Поля класса должны включать базовые и производные поля, поля с единичным и множественным значениями, а также статическое поле. Снабдите поля префиксами видимости.

11.2. Дополните модель класса `Patient`, разработанную при выполнении упражнения 11.1, OCL-ограничениями, уточняющими сам класс, производные поля и начальные значения полей.

11.3. Дополните модель класса `Patient`, разработанную при выполнении упражнения 11.1, стандартными get- и set-методами. Снабдите поля и методы префиксами видимости.

11.4. Дополните модель, разработанную при выполнении упражнения 11.3, OCL-ограничениями, уточняющими все get-методы, кроме тех, которые работают с полями с множественными значениями.

11.5. Дополните модель, разработанную при выполнении упражнения 11.3, одним нестандартным методом и уточните его при помощи OCL-ограничений.

11.6. Разработайте фрагмент кода модели класса `Patient`, полученной при выполнении упражнений 11.1 — 11.4. Код должен учитывать нарушение предусловий get-методов и не включать нестандартный метод, а также методы-конструкторы.

11.7. Дополните код, разработанный при выполнении упражнения 11.6, конструктором с параметрами.

11.8. Разработайте атрибутивную модель класса с именем `Clock`, моделирующего часы с секундным индикатором и функцией будильника. Снабдите класс и его поля всеми известными вам ограничениями.

11.9. Дополните модель, разработанную при выполнении упражнения 11.8, стандартными и нестандартными методами. Методы должны моделировать все функции часов. Снабдите один из нестандартных методов OCL-ограничениями.

11.10. Разработайте фрагмент кода модели класса `Clock`, полученной при выполнении упражнений 11.8 и 11.9. Код должен включать один конструктор, инициализирующий все поля класса, и один нестандартный метод. Код не должен включать стандартные get- и set-методы.

11.11. Предложите модель класса треугольников и введите в нее перегруженные методы. Включите в модель только те элементы, которые необходимы для иллюстрации перегруженных методов.

11.12. Предложите модель системы, состоящей из одного суперкласса с именем `Queue`, моделирующего любую очередь, и два подкласса, моделирующих очереди, упорядоченные в соответствии с правилами `FIFO` и `LIFO`. Система должна быть примером использования абстрактных методов и классов.

11.13. Разработайте модель и код интерфейса для класса, моделирующего стиральную машину-автомат.

11.14. Дополните модель класса `Clock`, полученную при выполнении упражнения 11.9, несколькими интерфейсами. Модель должна включать описания полей и методов класса. Интерфейсы должны быть заданы только своими именами (см. рис. 11.41). Покажите, каким образом эта модель отображается в код.

11.15. Для стандартных get- и set-методов класса `Person` (см. рис. 11.27), обеспечивающих доступ к одному из базовых полей, предложите OCL-ограничения, формулирующие условия доступа к этим полям.

Раздел 12

# МОДЕЛИРОВАНИЕ ПРОСТРАНСТВЕННОЙ СТРУКТУРЫ ПРИ ПОМОЩИ ДИАГРАММЫ КЛАССОВ

Пространственная структура программной системы моделируется при помощи ряда диаграмм, среди которых одной из наиболее важных является *диаграмма классов*. При помощи диаграммы классов моделируется *логическая структура* системы, которая остаётся неизменной на протяжении всего времени существования системы.

Диаграмма классов содержит информацию о том, из каких классов состоит система и в каких отношении они находятся друг с другом. Таким образом, диаграмма классов представляет собой множество графических символов классов и соединяющих их графических символов отношений.

В предыдущем разделе мы научились моделировать структуру отдельного класса. Теперь для того, чтобы получить навыки моделирования структуры системы, состоящей из нескольких классов, необходимо, в первую очередь, изучить вопросы, касающиеся моделирования отношений между классами.

Для моделирования отношений между классами UML предлагает набор типовых отношений, имеющих следующие наименования:

- *обобщение-специализация,*
- *ассоциация,*
- *композиция,*
- *агрегация,*
- *зависимость и*
- *реализация.*

Отношение типа обобщение-специализация используется для моделирования отношений между иерархически организованными классами или интерфейсами, использующими механизм наследования.

Отношение типа ассоциация используется для моделирования отношения между классами в том случае, когда объекты ассоциированных классов объединяются в новую сущность.

Отношения типа композиция и агрегация моделируют отношение между классом, рассматриваемым как целое, и классами, рассматриваемыми как части целого.

Отношение типа зависимость показывает, как один класс зависит от другого, либо какая существует взаимозависимость между классами.

Отношение типа реализация используется для того, чтобы показать отношение между интерфейсом или группой интерфейсов и классом, который реализует эти интерфейсы.

## 12.1. Отношение типа обобщение-специализация

Отношение типа обобщение-специализация позволяет моделировать иерархию классов и интерфейсов. Обобщение происходит при движении вверх по дереву иерархии (от подклассов к суперклассу), а специализация — при движении вниз по дереву иерархии (от суперкласса к подклассам).

Например, класс красных автомобилей может быть разделён или специализирован на подкласс красных автомобилей с двумя дверьми и подкласс красных автомобилей с четырьмя дверьми. При переходе от класса красных автомобилей к классу красных автомобилей с двумя дверьми осуществляется специализация, а при переходе в обратном направлении — обобщение.

*Графический символ отношения типа обобщение-специализация* представляет собой линию с полым треугольником на одном из её концов, как это было показано ранее в разделах 9 и 11. Треугольник указывает на направление обобщения. Специализация осуществляется в противоположном направлении.

Отношение типа обобщение-специализация чаще всего используется для моделирования *иерархии наследования*. Диаграммы классов, приведенные в предыдущих разделах, иллюстрируют структуру иерархии наследования для случая *одиночного наследования*. При одиночном наследовании каждый подкласс наследует члены только у одного суперкласса. Полые треугольники на этих рисунках *указывают направление наследования*. На рис 12.1 приведены две диаграммы классов, моделирующие иерархию одиночного наследования и иллюстрирующие два способа изображения графического символа отношения типа обобщения-специализации.

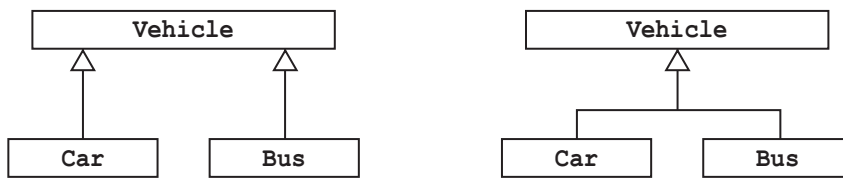



**Рис. 12.1.** Два способа изображения графического символа отношения типа обобщение-специализация

В обоих случаях суперкласс `Vehicle` (транспортное средство) находится в отношении обобщение-специализация с двумя подклассами: `Car` (автомобиль) и `Bus` (автобус). Несмотря на то, что оба приведенные на рис. 12.1 способа изображения графического символа отношения обобщение-специализация имеют одинаковый смысл и семантически эквивалентны, чаще используется способ, приведенный в правой части рис. 14.1, поскольку, как это будет видно из последующего изложения, он упрощает специфицирование ограничений для множества подклассов.

На рис. 12.2 приведен пример диаграммы классов, моделирующей иерархию наследования для случая *множественного наследования*. При множественном наследовании подкласс может наследовать поля и/или методы у нескольких суперклассов.

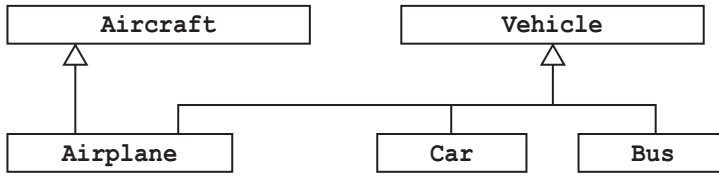



**Рис. 12.2.** Пример моделирования множественного наследования

Диаграмма классов на рис. 12.2 моделирует структуру, в которой класс `Airplane` (самолёт) является подклассом сразу двух классов: `Aircraft` (летательный аппарат) и `Vehicle` (транспортное средство). Таким образом, класс `Airplane`, кроме собственных полей и методов, может быть «собран» из полей и методов двух суперклассов.

Механизм наследования предполагает два способа формирования подклассов из имеющегося суперкласса:

- путём *расширения* атрибутивной модели и/или поведения суперкласса;
- путём *переопределения* атрибутивной модели и/или поведения суперкласса.

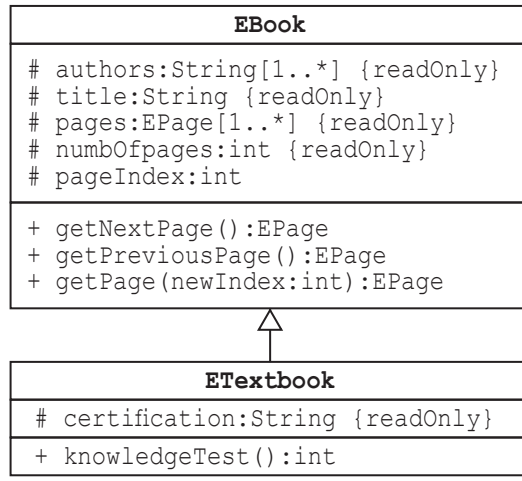
### 12.1.1. Расширение суперкласса

Способ формирования подкласса путём расширения суперкласса соответствует определению и описанию механизма наследования, рассмотренного в подразделе 9.1.6.

Списки полей и методов в суперклассе моделируют наиболее общие атрибуты и поведение некоторого класса объектов, а списки полей и методов его подклассов моделируют специфические атрибуты и специфическое поведение подклассов. С учётом механизма наследования атрибутивная модель и поведение подкласса (тип подкласса) всегда «шире» атрибутивной модели и поведения суперкласса (типа суперкласса) в том смысле, что списки полей и методов подклассов расширены по

отношению к спискам полей и методов суперкласса. Следствием отмеченного соотношения между типом суперкласса и типами его подклассов является принцип подстановки Барбары Лисков. *Принцип подстановки Лисков* означает, что в объектной системе объекты некоторого класса могут быть заменены на объекты любого из его подклассов без нарушения корректности программы.

Проиллюстрируем способ формирования подклассов путём расширения суперкласса на примере. Рассмотрим систему, состоящую из класса `EBook` (электронная книга) и его подкласса `ETextbook` (электронный учебник). Модель пространственной структуры этой системы приведена на рис. 12.3.




```
context EBook::pageIndex:int
init: 1
-- начальное значение индекса указывает на первую страницу

context EBook::getNextPage():EPage
pre:  pageIndex < numbOfPages
post: pageIndex = pageIndex + 1

context EBook::getPreviousPage():EPage
pre:  pageIndex > = 1
post: pageIndex = pageIndex - 1

context EBook::getPage(newIndex:int):EPage
pre:  newIndex > 1 and newIndex < numbOfPages
post: pageIndex = newIndex
```

**Рис. 12.3.** Модель, иллюстрирующая формирование подкласса путём расширения суперкласса

Класс EBook моделирует некоторые общие свойства и поведение электронной книги. Атрибутивная модель этого класса включает следующие поля: authors (авторы); title (заголовок), pages (страницы), numbOfPages (количество страниц) и pageIndex (номер текущей страницы). Страница имеет свою внутреннюю структуру, определяемую типом EPage (электронная страница). Поля класса EBook являются базовыми, нестатическими и определяемыми в классе. Все поля, кроме поля pageIndex, снабжены ограничением {readOnly}, и поэтому их значения можно только читать. Все поля снабжены префиксом видимости protected, и поэтому они доступны как методам класса EBook, так и методам его подкласса ETextbook.

Поведение класса EBook моделируется тремя методами-запросами: getNextPage (получить следующую страницу), getPreviousPage (получить предыдущую страницу) и getPage (получить произвольную страницу по её номеру newIndex). Нормальная работа перечисленных методов регламентируется пред- и постусловиями.

Перед выполнением метода getNextPage необходимо убедиться в том, что текущий номер страницы не соответствует последней странице. После выполнения метода getNextPage необходимо убедиться, что номер текущей страницы увеличен на единицу.

Перед выполнением метода getPreviousPage проверяется номер текущей страницы. Метод не может выполняться, если номер текущей страницы соответствует первой странице. После выполнения метода getPreviousPage необходимо убедиться, что номер текущей страницы уменьшен на единицу.

Перед выполнением метода getPage проверяется, находится ли номер страницы, переданный этому методу в качестве значения параметра, в диапазоне между единицей и номером последней страницы, а после выполнения этого метода необходимо проверить, что фактическое значение параметра метода getPage стало номером текущей страницы.

Класс EBook расширен подклассом ETextbook (электронный учебник). Наследование из класса EBook в класс ETextBook моделируется графическим символом отношения типа обобщение-специализация. Класс ETextbook расширяет атрибутивную модель класса EBook путём добавления поля certification (сертификат) — документа, специфического для учебников. Класс ETextbook расширяет поведение класса EBook путём добавления метода knowledgeTest (тест знаний), используемого для проверки и оценивания знаний студентов, работающих с электронным учебником. При отображении модели, приведенной на рис 12.3, в код, необходимо записать код обоих классов. Факт наличия между классами отношения типа обобщение-специализация фиксируется в заголовке подкласса при помощи служебного слова extends (расширяет). На рис. 12.4.приведен пример отображения модели на рис 12.3 в код. Служебное слово extends в заголовке класса ETextbook индицирует, что он является подклассом класса EBook. В коде опущены конструкторы обоих классов.

```
class EBook {
   // описание полей
   protected final String[] authors;
   protected final String title;
   protected final EPage[] pages;
   protected final int numbOfPages;
   protected int pageIndex;
   // описание методов
   public EPage getNextPage() {
      // код метода getNextPage, учитывающий
      // пред- и постусловия
   }
   public EPage getPreviousPage() {
      // код метода getPreviousPage, учитывающий
      // пред- и пост-условия
   }
   public EPage getPage(newIndex:int) {
      // код метода getPage, учитывающий
      // пред- и постусловия
   }
}
class ETextbook extends EBook {
   // описание полей
   protected final String certification;
   // описание методов
   public int knowledgeTest() {
      // код метода knowledgeTest
   }
}
```

**Рис. 12.4.** Отображение в код модели, приведенной на рис. 12.3

### 12.1.2. Переопределения членов суперкласса

Второй способ формирования подклассов заключается в том, что *в подклассе переопределяются унаследованные члены суперкласса*.

Переопределение поля означает, что в подклассе объявляется поле с тем же именем, что и в суперклассе, а переопределение метода означает, что в подклассе объявляется метод с тем же заголовком, что и в суперклассе, но с новым кодом. Таким образом, переопределение полей и методов в подклассе приводит к тому, что *в объекте подкласса появляется несколько пар одноимённых полей и/или методов*. Например, объект подкласса может содержать метод `m`, который

он наследует из суперкласса, а также собственный переопределённый метод `m`. Наличие одноимённых полей либо методов в объекте подкласса порождает проблему выбора одного из них в случае обращения извне. Например, если объект подкласса получает сообщение, вызывающее упомянутый метод `m`, то возникает вопрос о том, какому из двух методов передать управление. В случае переопределения выбор производится по следующему правилу. Если обращение к переопределённому полю/методу осуществляется по его имени/сигнатуре, то доступ обеспечивается к полю или методу, переопределённому в подклассе, а не к полю или методу, изначально определённому в суперклассе. Однако, доступ к исходным полям и методам, определённым в суперклассе, также возможен, если при обращении к ним указывать составное имя, включающее ссылку на объект суперкласса.

В том случае, когда в подклассе переопределено некоторое поле суперкласса, часто говорят не о переопределении, а о сокрытии поля. Когда в подклассе появляется поле с тем же именем, что и в суперклассе, то прежнее продолжает существовать, но перестаёт быть доступным, если обращаться к нему непосредственно по имени из метода подкласса.

Рассмотрим пример, иллюстрирующий формирование подкласса путём переопределения членов суперкласса. Предположим, что разработчики электронного учебника, пример которого мы использовали в предыдущем параграфе, разработали улучшенную и более эффективную версию теста знаний, моделируемого на рис. 12.3 методом `knowledgeTest`, и хотят использовать её в новых версиях учебника. Каким образом разработать новую версию программы с минимальными изменениями существующей версии? Хороший ответ на этот вопрос может быть следующим. Необходимо в систему, изображённую на рис. 12.3, ввести ещё один класс, например, `ETextbookImproved`, являющийся подклассом `ETextbook`, и переопределить в нём метод `knowledgeTest`. В этом случае при вызове метода `knowledgeTest` в объекте класса `ETextbookImproved` всегда будет работать новая версия метода `knowledgeTest`. Диаграмма классов, моделирующая структуру образовавшейся системы, приведена на рис. 12.5.

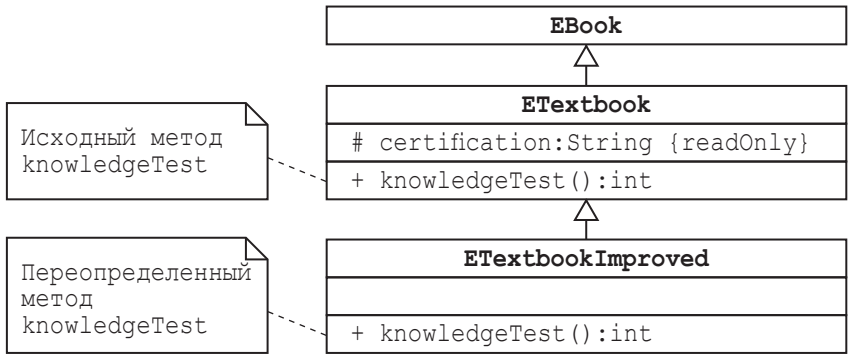



**Рис. 12.5.** Пример формирования подкласса путём переопределения членов суперкласса

Диаграмма классов на рис. 12.5 иллюстрирует относительность понятий суперкласса и подкласса. Класс `ETextbook` является подклассом для класса `EBook` и одновременно суперклассом для класса `ETextbookImproved`.

На рис. 12.6 приведен Java-код, соответствующий модели на рис. 12.5. В программе объявлен ещё один класс с именем `ETextbookImproved`, и в его заголовке указано, что он является подклассом класса `ETextbook`.

```
class EBook {
  . . .
}
class ETextbook extends EBook {
   protected final String certification;
   public int knowledgeTest() {
      // код изначально определённого метода knowledgeTest
   }
}
class EtextbookImproved extends ETextbook {
   public int knowledgeTest() {
      // код переопределённого метода knowledgeTest
   }
}
```

**Рис. 12.6.** Отображение в код модели, приведенной на рис. 12.5

## 12.2. Декомпозиция суперкласса и ограничения декомпозиции

С точки зрения степени подробности описания отношения типа обобщение-специализация, приведенные ранее диаграммы классов отражают только информацию о том, какие из классов в иерархии классов имеют статус суперкласса, а какие — статус подкласса. Часто при изображении иерархии классов полезна дополнительная информация, характеризующая всё множество подклассов. Например, является ли множество подклассов на рис. 12.1 (`Car` и `Bus`) полным или имеются ещё какие-то подклассы, для которых класс `Vehicle` может рассматриваться как суперкласс.

Множество подклассов будем называть также множеством декомпозиции суперкласса. Это наименование отражает взгляд на иерархию наследования как на некоторое классификационное дерево. В этом смысле суперкласс разделяется на множество подклассов, каждый из которых, в свою очередь, может быть разделён на множество подклассов и т. д. Декомпозиция, таким образом, понимается как разделение класса на подклассы.

В UML имеется две пары альтернативных ограничений, при помощи которых можно уточнить диаграмму классов и специфицировать общие свойства множества декомпозиции. Эти альтернативные ограничения называются:

- disjoint (не совмещённое) или overlapping (перекрывающееся);
- *incomplete (не полное)* или *complete (полное)*.

Рассмотрим смысловое значение этих ограничений на примерах. На рис. 12.7 изображена система, включающая суперкласс `Vehicle` (транспортное средство) и его три подкласса: `Airplane` (самолёт), `Car` (автомобиль) и `Horse` (лошадь). Диаграмма содержит дополнительную информацию о подклассах в виде ограничений `{disjoint}` и `{incomplete}`. Ограничение `{disjoint}` означает, что множество подклассов является не совмещённым и, следовательно, что среди объектов этих подклассов не найдётся ни одного, который обладал бы атрибутами и поведением объектов нескольких подклассов. Говоря иными словами, каждый из классов описывается своим уникальным набором полей и методов. Ограничение `{incomplete}` означает, что множество подклассов не полное и, следовательно, оно может быть дополнено некоторым количеством других подклассов. Для примера, приведенного на рис. 12.7, множество подклассов может быть дополнено, например, классом `Truck` (грузовик).

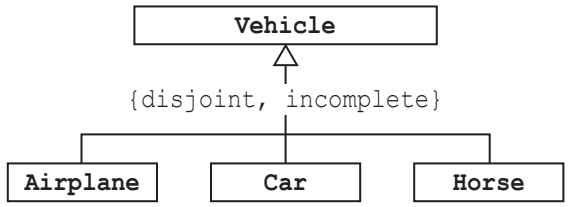



**Рис. 12.7.** Пример иерархии классов с ограничениями `{disjoint}` и `{incomplete}`

В ряде случаев подклассы перекрываются (содержат перекрывающееся множество объектов), и тогда множество декомпозиции может быть уточнено ограничением `{overlapping}`. На рис. 12.8 приведен пример, иллюстрирующий случай применения ограничения `{overlapping}`.

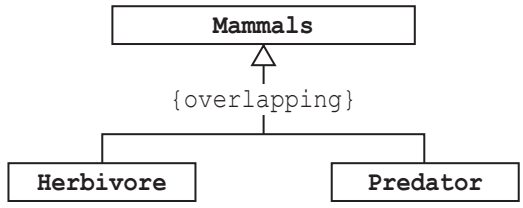



**Рис. 12.8.** Пример иерархии классов с ограничением `{overlapping}`

Класс `Mammals` (млекопитающие) распадается на два подкласса `Herbivore` (травоядные) и `Predator` (хищники). Ограничение `{overlapping}` означает, что

множество подклассов является перекрывающимся и, следовательно, существуют объекты, атрибуты которых принадлежат обоим подклассам (млекопитающие, которые питаются и растительной, и животной пищей).

Если множество декомпозиции суперкласса специфицировано при помощи ограничения `{overlapping}`, то применимо следующее правило о возможном развитии пространственной структуры системы. *Если известно, что классы перекрываются, то в модель системы можно включить ещё один подкласс, который наследует члены перекрывающихся классов*.

Рис. 12.9 показывает возможное развитие диаграммы классов, приведенной на рис. 12.8, в соответствии с приведенным выше правилом. Как видно на рис. 12.9, в структуру системы введен класс `Omnivore` (всеядное), наследующий поля и методы у классов `Herbivore` и `Carnivore`.

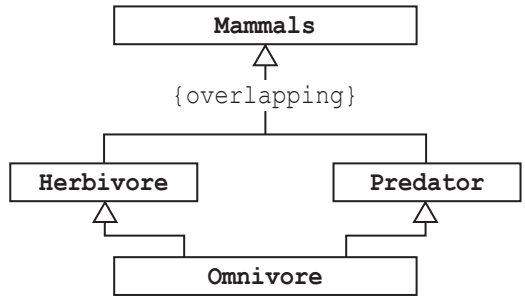



**Рис. 12.9.** Развитие диаграммы с ограничением `{overlapping}`

Если множество декомпозиции суперкласса снабжено ограничением `{complete}`, то этот факт позволяет сформулировать ещё одно правило о возможном развитии структуры системы. *Если известно, что множество подклассов полное, то их общий суперкласс можно представить в виде абстрактного класса.* Суперкласс, который разделяется на полное множество подклассов, можно объявить как абстрактный класс, поскольку при полном разделении суперкласса нет необходимости создавать объекты с его помощью. *Любой объект, который мог бы быть создан при помощи суперкласса, может быть создан при помощи одного из подклассов полного множества подклассов.* В левой части рис. 12.10 приведена диаграмма классов с ограничением декомпозиции `{complete}`, а в правой части — её возможное развитие.

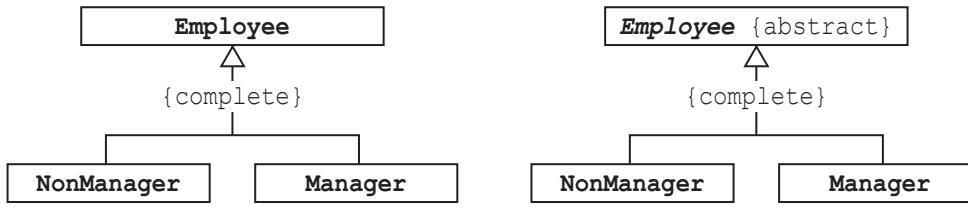



**Рис. 12.10.** Развитие диаграммы с ограничением `{complete}`

На рис. 12.10 суперкласс `Employee` (наёмный работник) может быть абстрактным, поскольку любой объект, моделирующий наёмного работника, может быть создан либо при помощи класса `NonManager` (не менеджер), либо при помощи класса `Manager` (менеджер).

При уточнении декомпозиции суперкласса при помощи ограничений не обязательно использовать оба ограничения. Количество и конкретный набор ограничений определяются целью моделирования.

## 12.3. Наследование интерфейсов

Отношение типа обобщение-специализация может быть установлено не только между классами, но и между интерфейсами. Поэтому справедливы понятия *суперинтерфейс* (супертип) и *подинтерфейс* (подтип). Диаграмма на рис. 12.11 иллюстрирует использование отношения типа обобщение-специализация для моделирования множественного наследования интерфейсов.

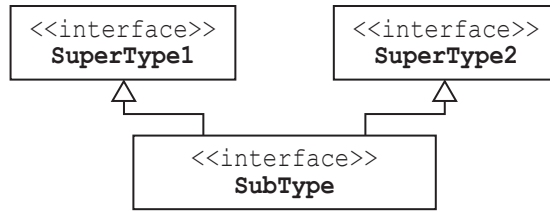



**Рис. 12.11.** Использование отношения типа обобщение-специализация для моделирования множественного наследования интерфейсов

На рис. 12.11 интерфейс `SubType` является подинтерфейсом, или расширенным интерфейсом. Расширенный интерфейс `SubType` получен путём расширения двух суперинтерфейсов: `SuperType1` и `SuperType2` или путём множественного наследования членов перечисленных интерфейсов. Отношение типа обобщение-специализация для случая интерфейсов легко отображается в программный код. Для этого используется знакомое нам служебное слово `extends`. После служебного слова `extends` располагается список имён суперинтерфейсов. Так, например, диаграмма, приведенная на рис. 12.11, отображается в Java-код следующим образом:

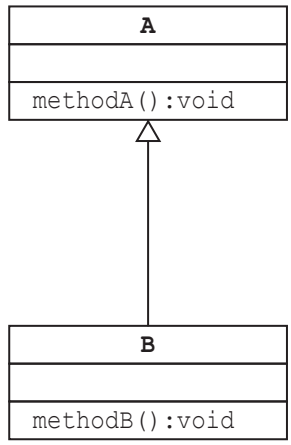
```
public interface Subtype extends SuperType1,SuperType2 {
   // . . .
}
```

Язык программирования Java допускает множественное наследование интерфейсов, но единичное наследование классов.

## 12.4. Наследование контракта

Наследование является одной из наиболее фундаментальных идей объектно-ориентированной парадигмы программирования, поэтому важным является ответ на вопрос о том, распространяется ли идея наследования и на контракт класса.

Общее правило заключается в том, что *контракт суперкласса полностью наследуется каждым из его подклассов*, а рис. 12.12 показывает, каким образом осуществляется наследование контракта.




Контракт класса A

```
context A
inv: <инвариант класса A>

context A::methodA():void
pre: <предусловие A
post: <постусловие A>
```

Контракт класса B

```
context B
inv: <инвариант класса B>

context B::methodB():void
pre: <предусловие B>
post: <постусловие B>
```

Расширенный контракт класса B

```
context B
inv: <инвариант класса A> and
     <инвариант класса B>

context B::methodA():void
pre: <предусловие A>
post: <постусловие A>

context B::methodB():void
pre: <предусловие B>
post: <постусловие B>
```

**Рис. 12.12.** Наследование контракта

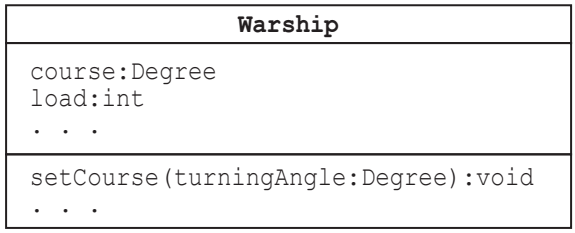
На рис. 12.12 приведена структура системы, состоящей из суперкласса A и его единственного подкласса B. Внутренняя структура обоих классов минимально проста и отражает только то, что необходимо для иллюстрации наследования контракта. Каждый из классов A и B уточнены при помощи контрактов, специфицирующих инварианты, а также предусловия и постусловия методов. Будем считать, что в контрактах обоих классов использованы различные OCL-выражения, специфицирующие инварианты, предусловия и постусловия.

Класс B наследует у класса A не только его поля и методы, но и контракт. Поэтому *контракт класса* B *является расширенным* и включает ограничения, задаваемые как контрактом суперкласса, так и контрактом подкласса.

Любой из объектов подкласса содержит как члены, унаследованные у суперкласса, так и члены, специфицированные в подклассе и, следовательно, должен одновременно удовлетворять инварианту, унаследованному у суперкласса, и инварианту, специфицированному в подклассе. Поэтому *расширенный контракт содержит единственный инвариант, полученный путем объединения инвариантов суперкласса и подкласса при помощи операции «и»*.

Методы methodA и methodB независимы, поэтому *в расширенный контракт класса* B *включены два OCL-ограничения, специфицирующие предусловия и постусловия обоих методов*.

Рассмотрим пример иерархии наследования, иллюстрирующий принципы наследование контракта. На рис. 12.13 приведена модель класса Warship (военный корабль).




Контракт класса Warship

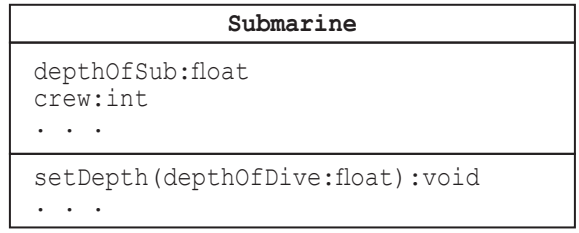
```
context Warship
inv: load < 500000

context Warship::setCourse(turningAngle:Degree):void
pre: turningAngle <= 300
post: course = course@pre + turningAngle
```

**Рис. 12.13.** Модель класса Warship

Атрибутивная модель класса Warship представлена двумя полями, которые моделируют наиболее общие атрибуты кораблей: course (курс) и load (загрузка).

Любой корабль движется некоторым курсом и загружен некоторым полезным грузом. Поведение объектов класса `Warship` моделируется методом `setCourse` (установить курс). Метод позволяет установить новый курс корабля, отличающийся от прежнего на величину, задаваемую входным параметром `turningAngle` (угол поворота).

Контракт класса `Warship` включает инвариант, который требует, чтобы полезный груз любого корабля этого класса не превышал 50 000 тонн. Контракт включает также пред- и постусловия метода `setCourse`. Смысл предусловия в том, чтобы ограничить угол поворота, который не должен превышать `300`. Постусловие проверяет, соответствует ли новый курс заданному курсу. На рис. 12.14 приведена модель класса `Submarine` (подводная лодка).




Контракт класса Submarine

```
context Submarine
inv: crew > 40 and crew < 50

context Submarine::setDepth(depthOfDive:float):void
pre: depthOfSub + depthOfDive <= 200
post: depthOfSub = depthOfSub@pre + depthOfDive
```

**Рис. 12.14.** Модель класса `Submarine`

Атрибутивная модель класса `Submarine` представлена двумя полями: `depthOfSub` (глубина погружения) и `crew` (экипаж). Любая подводная лодка может плыть под водой и управляется экипажем. Поведение объектов класса `Submarine` моделируется методом `setDepth` (установить глубину). Метод позволяет установить новую глубину погружения подводной лодки, отличающуюся от предыдущей глубины на величину, задаваемую входным параметром `depthOfDive` (глубина ныряния).

Контракт класса `Submarine` содержит инвариант, который требует, чтобы количество членов экипажа находилось в диапазоне от 40 до 50 человек. В контракт включены пред- и постусловия метода `depthOfDive`. Смысл предусловия в том, чтобы ограничить глубину погружения. Она не должна превышать 200 метров. Постусловие проверяет, соответствует ли новая глубина погружения заданной глубине.

Класс Submarine целесообразно рассматривать как подкласс класса Warship, поскольку в этом случае объекты класса Submarine могут наследовать поля и методы, определенные в классе Warship. Поскольку оба класса снабжены контрактами, то в том случае, когда класс Submarine получает статус подкласса, он, кроме полей и методов, наследует также и контракт класса Warship. На рис. 12.15 приведен расширенный контракт класса Submarine.

Расширенный контракт класса Submarine

```
context Submarine
inv: load < 500000 and
     crew > 40 and crew < 50

context Submarine::setDepth(depthOfDive:float):void
pre: depthOfSub + depthOfDive <= 200
post: depthOfSub = depthOfSub@pre + depthOfDive

context Submarine::setCourse(turningAngle:Degree):void
pre: turningAngle <= 300
post: course = course@pre + turningAngle
```

**Рис. 12.15.** Расширенный контракт класса Submarine

Как видно на рис. 12.15, в расширенном контракте класса Submarine имеется один инвариант, сформированный при помощи операции «и» из собственного инварианта класса Submarine и инварианта, унаследованного у класса Warship. Расширенный контракт класса Submarine включает пред- и постусловия собственного метода setDepth, а также пред- и постусловия метода setCourse, унаследованного у класса Warship.

## 12.5. Отношение типа ассоциация

Ранее, при изучении полей, мы ввели понятие отношения типа ассоциация, поскольку это было необходимо для различения полей, определяемых в символе класса и полей, определяемых ассоциацией. Целью настоящего подраздела является более подробное изучение отношения типа ассоциация.

Рассмотрим пример. Пусть имеется два класса независимых объектов: (1) класс Student, моделирующий студентов университета, и (2) класс Book, моделирующий книги, хранящиеся в университетской библиотеке. Если какой-либо студент берёт в библиотеке несколько книг во временное пользование, то объекты отмеченных классов перестают быть независимыми и *образуют новую сущность, представляющую*

*собой ассоциацию студента с некоторым количеством книг*. Таблица на рис. 12.16 иллюстрирует новые сущности, образовавшиеся в результате ассоциации объектов классов `Student` и `Book`.

На рис. 12.16 объекты класса `Studen` представлены именами студентов (левая часть рисунка), а объекты класса `Book` — наименованиями книг (правая часть рисунка). Связи, образующие новые сущности в виде студента и некоторого количества библиотечных книг, представлены средней частью рисунка. Отношение типа ассоциация можно рассматривать как совокупность отмеченных связей. На рис. 12.16 этому отношению присвоено имя `Borrow` (брать во временное пользование).

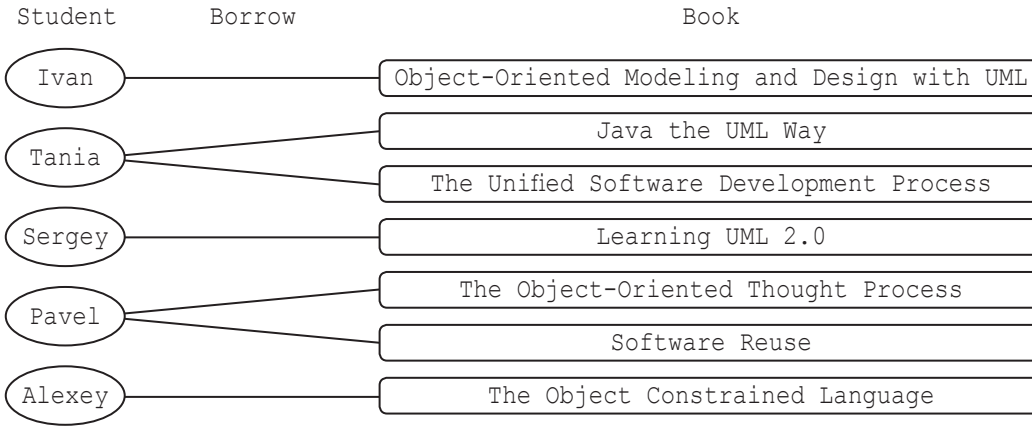



**Рис. 12.16.** Пример ассоциации объектов классов `Student` и `Book`

Приведенный пример иллюстрирует тот специфический характер отношения между классами, для моделирования которого необходимо использовать отношение типа ассоциация. Пример позволяет также сделать заключение относительно применимости отношения типа ассоциация. Отношение типа ассоциация используется в диаграмме классов в тех случаях, когда ранее независимые объекты различных классов ассоциируются (объединяются) в новые сущности путём установления связей между ними. В дальнейшем будет показано, что ассоциироваться в новые сущности могут не только объекты различных классов, но и объекты одного и того же класса.

Связи, при помощи которых ассоциируются объекты, неодинаковы и имеют различную природу. Например, каждая новая сущность, определяемая на рис. 12.16 ассоциацией `Borrow`, объединяет *одного студента* (один объект класса `Studen`) и *несколько книг* (несколько объектов класса `Book`). Такой характер связей, моделируемых ассоциацией `Borrow`, является следствием правила, согласно которому библиотеки разрешают одному студенту брать во временное пользование несколько книг, однако не разрешают нескольким студентам брать

во временное пользование одну и ту же книгу. Если мы рассмотрим пример ассоциации `Family` (семья) для объектов классов `Man` (мужчина) и `Woman` (женщина), то эта ассоциация образует новые сущности путём связывания *одного мужчины* (один объект класса `Man`) и *одной женщины* (один объект класса `Woman`), в соответствии с правилом, запрещающим нескольким женщинам вступать в брак с одним мужчиной, а нескольким мужчинам вступать в брак с одной женщиной.

### 12.5.1. Две нотации для отношения типа ассоциация

Существуют две нотации для изображения отношения типа ассоциация на диаграмме классов. Графическим символом ассоциации для обоих нотаций является *отрезок прямой*, соединяющий классы, а отличие заключается в способе записи имени ассоциации.

Диаграмма классов на рис. 12.17 иллюстрирует первую из этих нотаций. Диаграмма моделирует систему из трёх классов: `Person` (личность), `Institution` (организация) и `Country` (государство), между которыми установлены три ассоциации.

Каждая *ассоциация может иметь имя* для её идентификации, поскольку между двумя классами можно установить несколько ассоциаций, и их нужно различать. Например, объекты класса `Person` при образовании новых сущностей с объектами класса `Institution` могут рассматриваться как поставщики, заказчики, наёмные работники и т. д.

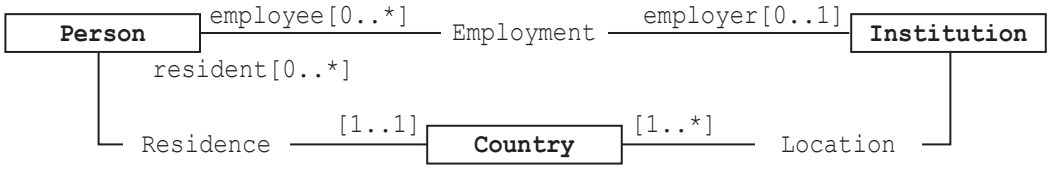



**Рис. 12.17.** Пример нотации для отношения типа ассоциация.
Имя ассоциации представлено в виде имени существительного

Имя ассоциации должно отражать семантику отношения и в рассматриваемой нотации записываться в виде имени существительного в единственном числе в разрыве графического символа ассоциации. Имя ассоциации не является обязательным, но если оно присутствует, то должно начинаться с большой буквы.

На рис. 12.17 между классами `Person` и `Institution` установлено отношение типа ассоциация с именем `Employment` (работа по найму), между классами `Person` и `Country` — отношение типа ассоциация с именем `Residence` (местожительство), а между классами `Country` и `Institution` — ассоциация с именем `Location` (местонахождение).

Мы уже знаем, что каждый из концов графического символа ассоциации называется *полюсом ассоциации*. На диаграмме классов полюса описываются именем полюса и множественностью полюса.

*Имя полюса* отражает роль объектов класса (к которому примыкает полюс) в новой сущности, которая образуется в результате ассоциации, а *множественность полюса* описывает возможное количество объектов класса (к которому примыкает полюс) в новой сущности. Полюс ассоциации `Employment` на стороне класса `Person` имеет имя `employee` (наёмный работник), а его множественность задаётся выражением `[0..*]`. Это выражение означает, что или ноль, или некоторое количество объектов класса `Person` могут входить в новые сущности, образуемые ассоциацией `Employment`. Полюс ассоциации `Employment` на стороне класса `Institution` имеет имя `employer` (работодатель) и множественность `[0..1]`. Это означает, что или ноль, или один объект класса `Institution` может входить в новые сущности, образуемые ассоциацией `Employment`.

Для специфицирования множественности полюса используются те же выражения, которые мы ранее использовали для специфицирования полей с множественными значениями. Эти выражения приведены на рис. 12.18.

`M..N` — от `M` до `N` (где, `M` и `N` целые положительные числа)
`0..1` — ноль или один
`1..1` — один и только один
`0..*` — от нуля до любого положительного целого
`1..*` — от единицы до любого положительного целого

**Рис. 12.18.** Выражения для специфицирования множественности полюса ассоциации

Имя ассоциации, а также имя и множественность полюса не являются обязательными элементами модели. В ряде случаев трудно придумать компактное имя полюса, отражающего его роль в ассоциации. Как видно на рис. 12.17, имена полюсов некоторых ассоциаций не указаны. Однако, если модель разрабатывается с целью дальнейшего кодирования, то необходимо описать полюса ассоциации, поскольку они, по сути, являются полями класса, определяемые ассоциацией.

Рассмотренная нотация для изображения ассоциации на диаграмме классов предполагает, что имя ассоциации записывается в виде имени существительного. Такая нотация удобна, когда в ходе дальнейшего развития диаграммы отношение типа ассоциация представляется в виде класса. Существует ещё одна нотация для изображения ассоциации. Эта нотация отличается от рассмотренной только тем, что имя ассоциации записывается не в виде имени существительного, а в виде глагола или глагольной группы.

Рис. 12.19 иллюстрирует второй способ изображения ассоциации на диаграмме классов.

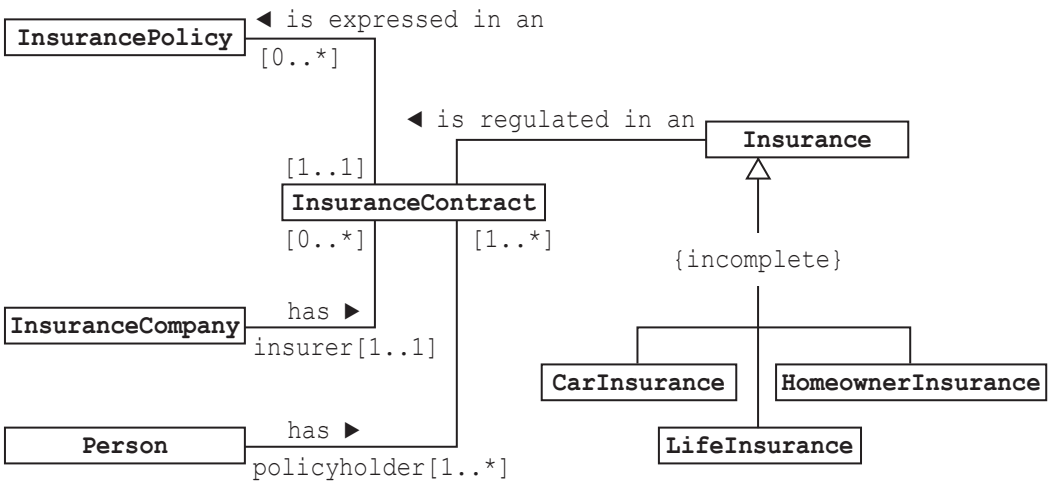



**Рис. 12.19.** Пример альтернативного способа изображения ассоциации на диаграмме классов. Имя ассоциации представлено в виде глагола

Способ изображения ассоциаций, приведенный на рис. 12.19, обладает тем достоинством, что позволяет составлять из имени ассоциации и имен классов, которые она объединяет, предложение, объясняющее смысл ассоциации. Маленький зачернённый треугольник показывает направление чтения этого предложения. Так, например, ассоциации, приведенные на рис. 12.19, образуют следующие предложения:

Insurance *is regulated in an* insurance contract;
Insurance contract *is expressed in an* insurance policy;
Insurance company *has* insurance contract;
Person *has* insurance contract.

На рис. 12.19 приведена модель структуры системы страхования некоторой страховой компании. Читая диаграмму, приведенную на рис. 12.19, можно получить, например, следующую информацию.

Личность может быть держателем страхового полиса (policyholder), а держатель страхового полиса обладает одним или многими страховыми контрактами.

Одному страховому контракту соответствует один или несколько держателей страхового полиса, которые, в свою очередь, являются личностями.

Одна страховая компания играет роль страховщика (insurer), который имеет ноль, один или много страховых контрактов с держателями страховых полисов.

Страховой контракт представляет страховку (insurance), которая может быть страховкой автомобиля (car insurance), страховкой жизни (life insurance) или страховкой недвижимости (homeowner's insurance).

Страховка регулируется страховым контрактом, который специфицирует держателя страхового полиса, срок страховки и стоимость страхового полиса. Страховой контракт включает эту информацию, моделируемую полями класса, в документе, называемом страховой полис.

Страховой контракт отображается в одном (или нулевом, если он ещё не изготовлен) страховом полисе.

Полюс ассоциации является способом представления поля, определяемого ассоциацией, которое хранит ссылку на объект. Этот факт позволяет трансформировать бинарное отношение типа ассоциации во внутренние поля классов, объединяемых ассоциацией, исключив из диаграммы графический символ ассоциации. На рис. 12.20 приведены две модели, иллюстрирующие это утверждение.

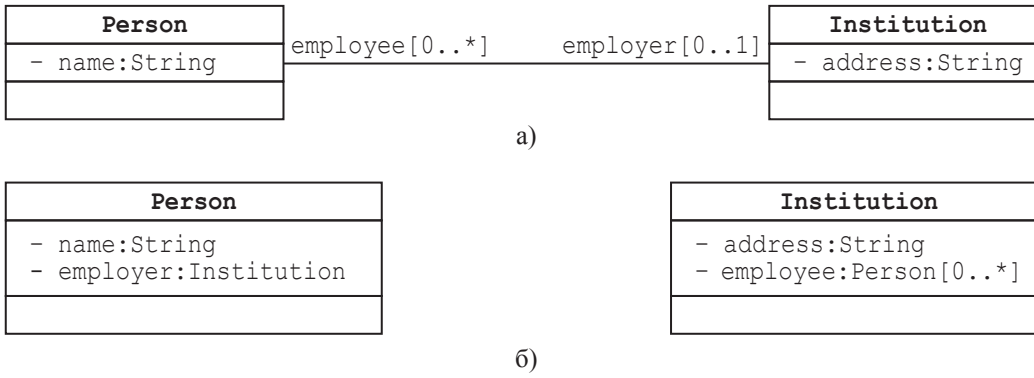



**Рис. 12.20.** Две модели, иллюстрирующие взаимную трансформацию полей, определяемых ассоциацией, и полей, определяемых в символе класса: а) модель с использованием отношения типа ассоциация; б) модель без использования отношения типа ассоциация

Модель, приведенная в верхней части рис. 12.20, является частью диаграммы классов, приведенной на рис. 12.17. В этой модели явно, при помощи отношения типа ассоциация, моделируются связи между объектами классов `Person` и `Institution`.

Модель, приведенная в нижней части рис. 12.20, представляет собой два класса, между которыми отсутствует отношение типа ассоциация. Однако, связи между объектами классов `Person` и `Institution` существуют. Они представлены неявно полями `employer` и `employee`, хранящими ссылки на объекты противоположного класса.

Возможность трансформации бинарного отношения типа ассоциация в поля, определяемые в символах классов, а также трансформации полей, специфицированных в символах класса, в отношение типа ассоциация между этими классами ставит вопрос о том, каким критерием следует пользоваться для определения местоположения поля в модели. Иными словами, как определить, должно ли поле принадлежать атрибутивной модели класса, или его следует определить при помощи ассоциации.

ООП утверждает, что все сущности окружающего нас мира могут представляться объектами, но не отвечает на вопрос о том, какие части мира должны представляться объектами. Например, в рамках ООП, такая сущность как человек в шляпе, может представляться различным образом: (1) как объект, одним из атрибутов которого, есть головной убор, являющийся шляпой; (2) как ассоциация двух объектов (объект-человек и объект-шляпа) и т. д. Возможно, что исследование этого вопроса позволит сформулировать некоторые фундаментальные принципы ООП, однако на практике ответ на на него, как правило, определяется опытом программиста.

При кодировании классов актуальность рассматриваемого вопроса о явном или неявном представлении ассоциации исчезает. В данном случае код на языке программирования Java является более грубым средствои описания программной сущности, чем UML-модель, поскольку в коде поле, определяемое в графическом символе класса, не отличается от поля, определяемого ассоциацией. У программиста есть только один способ связать объекты двух различных классов в новую сущность — при помощи перекрестных ссылок. Это означает, что в каждом из классов необходимо объявить поле для хранения ссылки/ссылок на объект/объекты противоположного класса. Поэтому обе модели, приведенные на рис. 12.20, отображаются не в различные, а в один и тот же код.

На рис. 12.21 приведен код, в который отображаются как модель, приведенная на рис. 12.20 а), так и модель, приведенная на рис. 12.20 б).

```
class Person {
   private String name;
   private Institution employer;
          . . .
}

class Institution {
   private String address;
   private Person[] employee;
          . . .
}
```

**Рис. 12.21.** Код классов `Person` и `Institution` для обеих моделей, приведенных на рис. 12.19

### 12.5.2. Представление ассоциации в виде класса

Отношение типа ассоциация используется для моделирования системы, в которой ранее независимые объекты одного или нескольких классов ассоциируются (объединяются) в новые сущности. Нотации, которые мы использовали для изображения ассоциации на диаграмме классов, представляли новые сущности в виде связей между

ассоциированными объектами. При этом модель отражала только факт наличия связей и никак их не описывала. Однако, модель может содержать более подробную информацию о связях, например, в виде их атрибутов (полей) и поведения (методов). Для этого необходимо рассматривать связи между объектами ассоциированных классов в качестве объектов класса, моделирующего отношение типа ассоциация.

Ничего не мешает нам мыслить ассоциацию как класс, объектами которого являются связи. Покажем, что такое рассмотрение ассоциации не только возможно, но и целесообразно. На рис. 12.22 приведен пример диаграммы классов с отношением типа ассоциация, имеющим имя `Fatherhood` (отцовство), которое установлено между классом `Man` (мужчина) и классом `Child` (ребёнок). Множественности полюсов этого отношения моделируют тот факт, что один мужчина может быть отцом нескольких детей, а один ребёнок может иметь только одного отца.

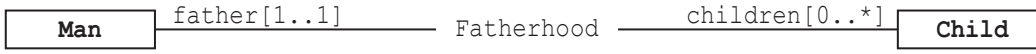



**Рис. 12.22.** Ассоциация «отцовство» между классами мужчин и детей

На рис. 12.23 приведен пример диаграммы классов, которая аналогична диаграмме, изображённой на рис. 12.22, но на этой диаграмме отношение `Fatherhood` представлено в виде класса.

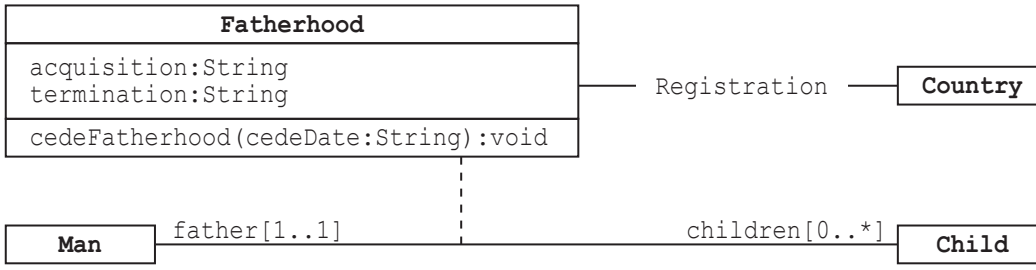



**Рис. 12.23.** Ассоциация `Fatherhood`, рассматриваемая как класс

При представлении ассоциации в виде класса в диаграмму вводится ещё один графический символ класса, который соединяется с графическим символом ассоциации при помощи пунктирной линии. Имя ассоциации становится именем этого класса. Поэтому, если предполагается дальнейшее развитие диаграммы с целью представления ассоциации в виде класса, то целесообразно использовать ту нотацию, в которой имя ассоциации задается в виде имени существительного.

Представление отношения типа ассоциация в виде класса означает, что мы *интерпретируем связи между объектами ассоциированных классов как объекты ассоциации-класса*. Такая интерпретация позволяет расширить и уточнить описание связей при помощи полей и методов ассоциации-класса, характеризующих

именно отношение ассоциации, а не ассоциированные классы. Например, в случае представления ассоциации `Fatherhood` в виде класса, можно уточнить это отношение следующими полями и методами:

`acquisition:String` — дата установления отношения отцовства (дата регистрации новорождённого ребёнка или дата регистрации усыновлённого ребёнка);

`termination:String` – дата прекращения отношения отцовства (дата регистрации смерти ребёнка или дата лишения мужчины родительских прав);

`cedeFatherhood(cedeDate:String)` — операция передачи права отцовства.

Представление отношения типа ассоциация в виде класса позволяет расширять модель не только путём введения полей и методов, описывающих ассоциацию, но и путём введения новых отношений типа ассоциация между ассоциацией-классом и другим классом или даже между двумя ассоциациями-классами. На рис. 12.23 введена ассоциация `Registration` (регистрация) между ассоциацией `Fatherhood` и классом `Country`, которая добавляет в модель сведения о том, в какой стране зарегистрировано отцовство.

Представление отношения типа ассоциация в виде класса изменяет структуру новой сущности, образуемой в результате ассоциации объектов. Покажем это на примере.

Пусть имеется объект класса `Man` с именем `Pavel` и несколько объектов класса `Child` с именами `Dasha` и `Masha`. В том случае, если упомянутые объекты классов `Man` и `Child` объединяются, образуя новую сущность, и мы моделируем образовавшуюся систему при помощи диаграммы классов, приведенной на рис. 12.22, то графически эту новую сущность можно представить в виде графа в верхней части рис. 12.24. Этот граф иллюстрирует тот факт, что новая сущность образована путём *непосредственного связывания* объекта `Pavel` (объект класса `Man`) с объектами `Dasha` и `Masha` (объекты класса `Child`).

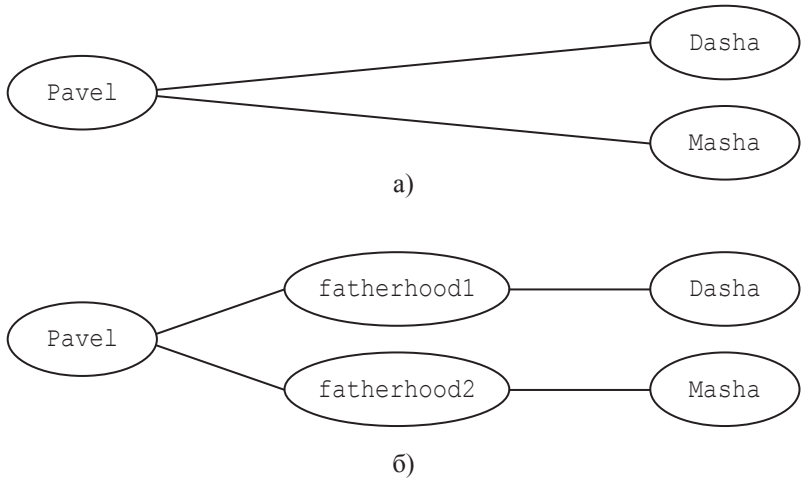



**Рис. 12.24.** Пример новой сущности, образуемой ассоциацией `Fatherhood`: а) ассоциация `Fatherhood` не представлена классом; б) ассоциация `Fatherhood` является классом

Если же мы представляем ассоциацию в виде класса и моделируем образовавшуюся систему при помощи диаграммы классов, приведенной на рис. 12.23, то графически структуру новой сущности можно представить в виде графа в нижней части рис. 12.24. Этот граф показывает, что объект `Pavel` (объект класса `Man`) связан с объектами `Dasha` и `Masha` (объекты класса `Child`) *опосредованно* — через объекты `fatherhood1` и `fatherhood2` (объекты класса `Fatherhood`).

### 12.5.3. Множественные, рекурсивные и тернарные ассоциации

Между двумя классами может быть установлено несколько отношений типа ассоциация. Так, например, между объектами класса личностей `Person` и объектами класса организаций `Institution` может быть установлена ассоциация, выражающая отношение между работодателями и наёмными работниками, и в то же время между этими классами может существовать ассоциация, выражающая отношение между заказчиками и исполнителями. Это различные отношения. В том случае, когда между двумя классами установлено несколько отношений типа ассоциация, их называют *множественными ассоциациями*. На рис. 12.25 приведен пример диаграммы классов, которая иллюстрирует множественные ассоциации между классами `SeaCraft` (морское судно) и `SeaPort` (морской порт).

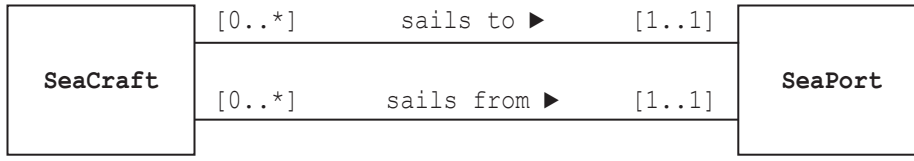



**Рис. 12.25.** Пример диаграммы классов с множественными ассоциациями

Ассоциация с именем `sails to` (плыть в) моделирует рейсы, осуществляемые в морской порт, а ассоциация `sails from` (плыть из) моделирует рейсы, осуществляемые из морского порта.

Приведенные ранее примеры диаграмм классов иллюстрируют бинарное отношение типа ассоциации, или отношение, объединяющее в новые сущности *объекты двух различных классов*. В общем случае, *отношение типа ассоциация может объединять в новые сущности объекты одного, двух или более классов*. Иными словами, отношение типа ассоциация может быть унарным, бинарным, тернарным и, в общем случае, n-арным.

Унарная ассоциация часто называется *рекурсивной ассоциацией*. Диаграмма классов, иллюстрирующая рекурсивную ассоциацию, приведена на рис. 12.26.

Ассоциация с именем `Family` (семья) моделирует семьи как объединения некоторого количества объектов одного и того же класса `Person` (личность). Из диаграммы, приведенной на рис. 4.26, следует, что среди объектов класса `Person` есть объекты, исполняющие роль родителей (`parents`), и объекты, исполняющие роль детей (`children`).

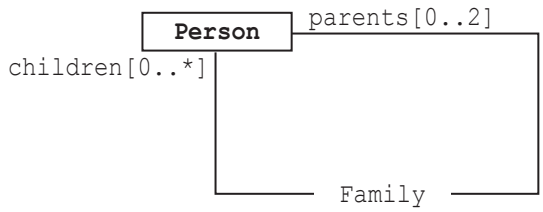



**Рис. 12.26.** Пример рекурсивной ассоциации

Множественности полюсов на рис. 12.26 интерпретируются следующим образом. Ребёнок может либо не иметь родителей, либо иметь одного или двух родителей, а родители могут либо не иметь ни одного ребёнка, либо иметь нескольких детей.

Между объектами одного и того же класса может быть установлено несколько рекурсивных ассоциаций.

Хотя бинарные и унарные (рекурсивные) ассоциации являются наиболее частотными, иногда возникает необходимость использовать *тернарные ассоциации*. Тернарная ассоциация нужна в ситуациях, когда новые сущности образуются путём объединения объектов, принадлежащих трём различным классам. На рис. 12.27 приведен пример тернарной ассоциации.

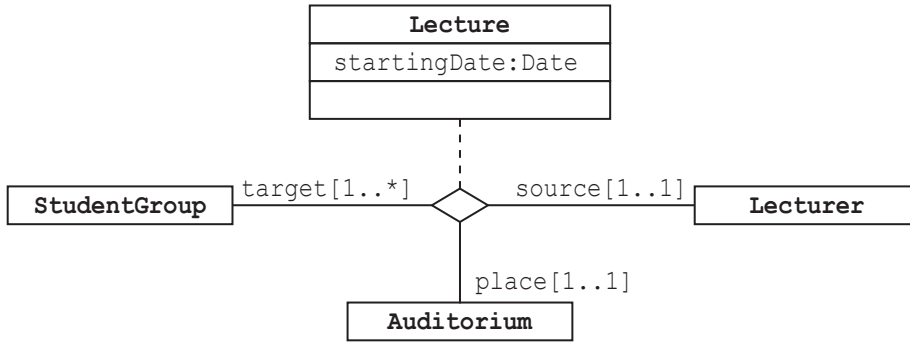



**Рис. 12.27.** Пример тернарной ассоциации

Диаграмма классов на рис. 12.27 моделирует лекцию (`Lecture`) в виде сущности, образованную путём связывания объектов трёх классов: `Lecturer` (лектор), `StudentGroup` (студенческая группа) и `Auditorium` (аудитория).

Для изображения тернарной ассоциации на диаграмме классов используется графический символ ромба, который объединяет графические символы ассоциаций, исходящие из символов классов, участвующих в ассоциации (левая, правая и нижняя вершины ромба на рис. 12.27), а также пунктирную линию, исходящую из символа класса, представляющего тернарную ассоциацию. Таким образом, тернарная ассоциация всегда представляется в виде класса. Это означает, что новые сущности,

образуемые тернарной ассоциацией, рассматриваются как объекты класса и могут быть описаны дополнительными полями и методами. На рис. 12.27 ассоциация-класс `Lecture` снабжена полем `startingDate` (дата начала лекций).

Тернарная ассоциация имеет три полюса, каждый из которых описывается именем и множественностью. На рис 12.27 полюс, примыкающий к классу `StudentGroup`, описан именем `target` (мишень) и множественностью `[1..*]`. Имя отражает роль объектов класса `StudentGroup` в ассоциации, а множественность — тот факт, что одна или несколько студенческих групп входят в состав новой сущности, моделируемой ассоциацией `Lecture`.

Полюс, примыкающий к классу `Lecturer`, описан именем `source` (источник) и множественностью `[1..1]`. Имя `source` отражает роль объектов класса `Lecturer` в новой сущности, а множественность означает, что только один объект класса `Lecturer` (один лектор) входит в состав новой сущности, моделируемой ассоциацией `Lecture`.

Полюс, примыкающий к классу `Auditorium`, описан именем `place` (место) и множественностью `[1..1]`. Имя `place` отражает роль объектов класса `Auditorium` в новой сущности, а множественность отражает тот факт, что только один объект класса `Auditorium` (одна аудитория) входит в состав новой сущности, моделируемой ассоциацией `Lecture`.

Если рассматривать ассоциацию объектов как способ создания новых сущностей, то ничего не мешает, по крайней мере, умозрительно, создавать новые сущности путём ассоциации объектов произвольного количества классов. Однако при практическом использовании отношения типа ассоциация в диаграммах классов обычно ассоциация объединяет не более трёх классов.

### 12.5.4. Навигация для отношения типа ассоциация

Навигацией для отношения типа ассоциация будем называть «знания» объекта одного из ассоциированных классов о том, какие объекты других классов входят в новую сущность, образуемую ассоциацией. Используя понятие связь, навигацию, в случае бинарной ассоциации, можно определить как «знания» объекта одного из ассоциированных классов о том, с какими объектами противоположного класса он связан.

Навигация, в случае бинарной ассоциации, может быть двусторонней и односторонней. Если ассоциация объединяет классы `A` и `B`, то *односторонняя навигация* означает, например, что объект класса `A` «знает», с какими объектами класса `B` он входит в новую сущность, образуемую ассоциацией, но объект класса `B` «не знает», с какими объектами класса `A` он ассоциирован, а *двусторонняя навигация* означает, что отмеченными знаниями обладают объекты обоих классов.

Графический символ для отношения типа ассоциация, который мы использовали ранее, не специфицировал направление навигации. Для указания направления навигации графический символ ассоциации снабжается дополнительными

элементами с указанием направления разрешенной навигации и направлением, в котором навигация запрещена. Проиллюстрируем использование навигации для отношения типа ассоциации на примере.

На рис. 12.28 приведены три одинаковые диаграммы классов с ассоциацией `Fatherhood` (см. рис. 12.22), отличающиеся направлением навигации.

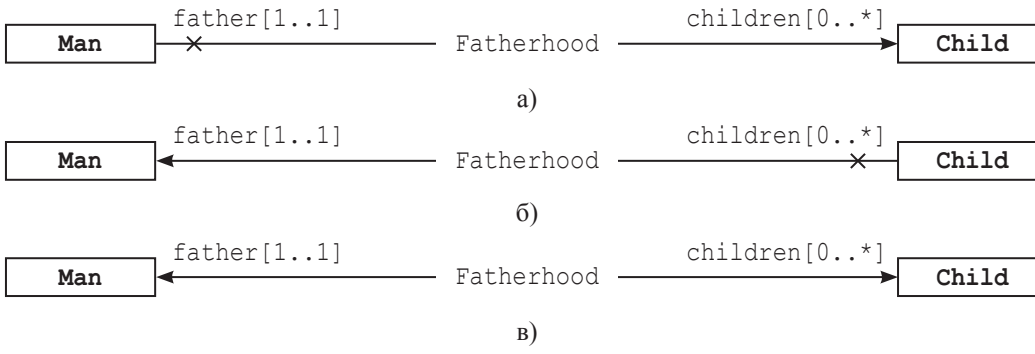



**Рис. 12.28.** Варианты навигации для ассоциации `Fatherhood`

Первый вариант навигации (рис. 12.28, а) означает, что для любого объекта класса `Man` имеет место навигация к объектам класса `Child`, связанных с этим объектом. Иными словами, для конкретного мужчины (объект класса `Man`), известны все его дети (объекты класса `Child`), но для конкретного ребёнка (объект класса `Child`) не известен его отец (объект класса `Man`). Графическим символом навигации, в случае односторонней навигации, является отрезок прямой, один конец которой снабжён стрелкой, а другой — крестиком. Стрелка указывает направление навигации, а крестик — на направление, в котором навигация запрещена. Ясно, что возможность первого варианта навигации должна быть обеспечена в классе `Man` полем с множественными значениями типа `Child`, хранящим ссылки на соответствующие объекты класса `Child`.

Второй вариант навигации (рис. 12.28, б) означает, что для любого объекта класса `Child` имеет место навигация к соответствующему объекту класса `Man`, связанному с этим объектом. Иными словами, для конкретного ребёнка (объект класса `Child`) известен его отец (объект класса `Man`), но для конкретного мужчины (объект класса `Man`) не известны его дети (объекты класса `Child`). Возможность второго варианта навигации обеспечивается полем типа `Man` в классе `Child`, хранящем ссылку на соответствующий объект класса `Man`.

Третий вариант (рис. 12.28, в) иллюстрирует двустороннюю навигацию и является комбинацией первых двух вариантов.

Варианты графического символа отношения типа ассоциация с учётом навигации, приведенные на рис. 12.28, наиболее частотны, однако они не исчерпывают все возможные варианты. Полный набор вариантов навигации приведен на рис. 12.29.

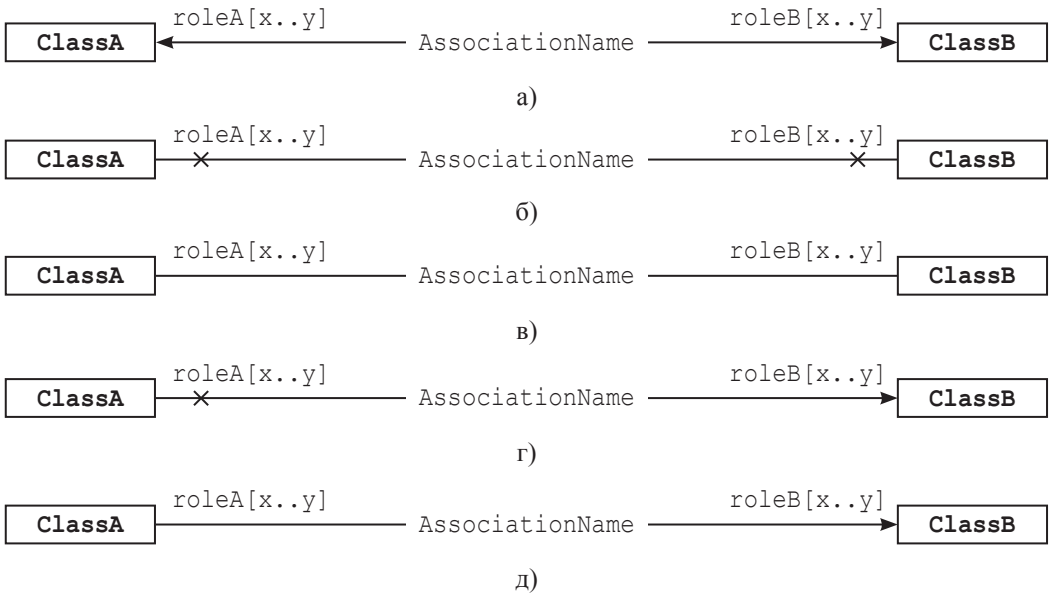
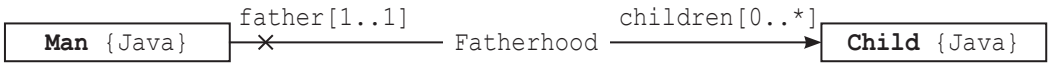



**Рис. 12.29.** Полный набор вариантов учёта навигации для отношения типа ассоциация: а) навигация возможна в обоих направлениях; б) навигация запрещена в обоих направлениях; в) навигация не специфицирована в обоих направлениях; г) навигация возможна в одном направлении и запрещена в противоположном; д) навигация возможна в одном направлении и не специфицирована в противоположном

### 12.5.5. Отображение бинарной ассоциации в программный код

Программный код, в который отображается бинарное отношение типа ассоциация, зависит от того, представлена ли ассоциация в виде класса или моделирует только тот факт, что объекты каким-то образом связаны между собой. Рассмотрим случай, когда отношение типа ассоциация моделирует только «пучок» связей между объектами. Назовём такую ассоциацию *ассоциация-связь*. При отображении ассоциации-связи в программный код необходимо обеспечить возможность навигации между объектами ассоциированных классов. Эта возможность реализуется полями, которые размещаются в классе, из которого должна быть обеспечена навигация. Поля могут иметь единичное или множественное значение, в зависимости от множественности полюса ассоциации на диаграмме классов. Имя полюса отображается в имя поля. Покажем, каким образом работают сформулированные правила при отображении в код моделей, приведенных на рис. 12.28.

На рис. 12.30 изображена первая из диаграмм классов, изображённых на рис. 12.28, и соответствующий ей код.

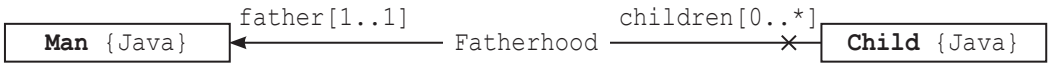



```
public class Man {
. . .
   private Child children[];
. . .
}

public class Child {
. . .
}
```

**Рис. 12.30.** Отображение отношения типа ассоциация-связь в программный код. Случай однонаправленной навигации и множественности полюса `[0..*]`

Как видно на рис. 12.30, в список полей класса `Man` добавлено поле, определяемое ассоциацией с именем `children`, имеющее тип `Child`. Поскольку множественность полюса `children` задана выражением `[0..*]`, то это поле описано в виде одномерного массива и содержит множество ссылок на объекты класса `Child`. Ясно, что введение отмеченного поля только в класс `Man` обеспечивает одностороннюю навигацию из класса `Man` в класс `Child`.

На рис. 12.31 приведена вторая из диаграмм классов, изображённых рис. 12.28, и соответствующий ей код.



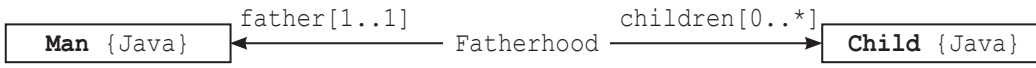

```
public class Man {
. . .
}

public class Child {
. . .
   private Man father;
. . .
}
```

**Рис. 12.31.** Отображение отношения типа ассоциация-связь в программный код. Случай однонаправленной навигации и множественности полюса `[1..1]`

В примере, приведенном на рис. 12.31, в список полей класса `Child` добавлено поле с именем `father` и типом `Man`. На диаграмме классов множественность полюса

father определяется выражением [1..1], поэтому поле father хранит единственную ссылку на объект класса Man и обеспечивает навигацию из класса Child в класс Man. Поскольку в классе Child отсутствуют ссылки на объекты класса Man, то навигация из класса Child в класс Man отсутствует.

На рис. 12.32 приведена третья из диаграмм классов рис. 12.24 и соответствующий ей код, являющийся комбинацией кодов, приведенных на рис. 12.30 и 12.31. Для обеспечения двусторонней навигации, поля, определяемые ассоциацией, добавлены как в код класса Man, так и в код класса Child.




```
public class Man {
. . .
   private Child children[];
. . .
}

public class Child {
. . .
   private Man father;
. . .
}
```

**Рис. 12.32.** Отображение отношения типа ассоциация-связь в программный код. Случай двунаправленной навигации

Рассмотрим теперь, каким образом отображается в код отношение типа ассоциация в том случае, когда это отношение представлено в виде класса. Назовём такую ассоциацию *ассоциация-класс*. В подразделе 12.5.2 мы отметили целесообразность представления ассоциации в виде класса и показали, каким образом на диаграмме классов изображается ассоциация-класс. Напомним, что для этой цели используется графический символ класса, который соединяется с графическим символом ассоциации при помощи пунктирной линии (см. рис. 12.23). Хотя такой способ изображения ассоциации-класса соответствует стандарту UML, его нельзя назвать удачным, поскольку пунктирная линия не несёт никакой смысловой нагрузки, кроме того, что показывает, какая именно ассоциация представлена в виде класса. Точно такой же смысл имеет пунктирная линия в символе комментария.

Более удачным будет такой способ изображения диаграммы классов, при котором ассоциация-класс явно ассоциирован с исходными классами. Рассмотрим в качестве примера ассоциацию Fatherhood на рис. 12.23. На рис. 12.33 приведена диаграмма классов, полученная из диаграммы на рис. 12.23 и моделирующая систему, состоящую из ассоциированных классов Man и Child.

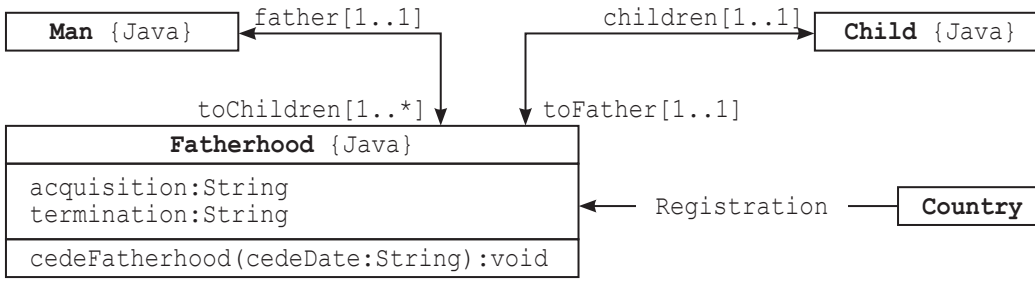



**Рис. 12.33.** Диаграмма классов, в которой ассоциация-класс `Fatherhood` явно ассоциирован с классами `Man` и `Child`

Множественности полюсов на диаграмме, изображённой на рис. 12.33, означают, что один объект класса `Man` связан с несколькими объектами класса `Fatherhood`, а один объект класса `Fatherhood` связан с одним объектом класса `Child`. При отображении диаграммы, приведенной на рис. 12.33, в программный код полюса ассоциаций транслируются в поля, определяемые ассоциацией в соответствующих классах с учётом того, что обе ассоциации имеют двустороннюю навигацию. На рис. 12.34 приведен код, соответствующий диаграмме, приведенной на рис. 12.33.

```
public class Man {
. . .
   private Fatherhood toChildren[];
. . .
}
public class Child {
. . .
   private Fatherhood toFather;
. . .
}
Public class Fatherhood {
. . .
   private String acquisition;  // определено в классе
   private String termination;  // определено в классе
   private Man father;          // определено ассоциацией
   private Child children;      // определено ассоциацией
   public void cedeFatherhood(String cedeDate);
   {
   // код метода cedeFatherhood
   }
}
```

**Рис. 12.34.** Отображение класса-ассоциации `Fatherhood` в программный код

В код класса `Man` введено поле с именем `toChildren` (к детям) и типом `Fatherhood`, хранящее ссылки на объекты класса `Fatherhood`, через которые обеспечивается навигация к соответствующим объектам класса `Child`. Аналогичное поле, с именем `toFather` (к отцу) и типом `Fatherhood`, введено в класс `Child`. Поле хранит ссылку на объект класса `Fatherhood`, через который обеспечивается навигация к соответствующему объекту класса `Man`. Поскольку ассоциация между классами `Man` и `Child` представлена классом, то её можно описать более подробно при помощи полей и методов, характеризующих ассоциацию, а не классы, между которыми установлено это отношение.

### 12.5.6. Отображение рекурсивной ассоциации в программный код

Рекурсивная ассоциация, так же, как и бинарная ассоциация, может быть ассоциацией-связью или ассоциацией-классом. Рекурсивная ассоциация-связь моделирует пучок связей между объектами одного и того же класса. При отображении рекурсивной ассоциации-связи в программный код необходимо учитывать следующие особенности.

Во-первых, несмотря на то, что на диаграмме классов рекурсивная ассоциация имеет два полюса, *только один из полюсов отображается в код*.

Во-вторых, *полюс ассоциации транслируется в поле, объявляемое рекурсивно*, и, следовательно, тип этого поля совпадает с именем класса.

В-третьих, рекурсивная ассоциация может иметь один или два варианта отображения в код. Если при помощи рекурсивной ассоциации моделируются связи между объектами, имеющими одинаковый ролевой статус (например, связи между объектами-узлами класса `Node` (узел), образующие сеть), то имеется только один вариант отображения модели в программный код. Если при помощи рекурсивной ассоциации моделируются связи между объектами, имеющими различный ролевой статус (например, связи между объектами-родителями и объектами-детьми для класса `Person` (личность), то имеется два варианта отображения модели в программный код.

Рассмотрим несколько примеров. Неориентированный граф можно представить в виде рекурсивной ассоциации объектов класса вершин (класс `Vertex`), поскольку рекурсивная ассоциация, по сути, моделирует отображение множества объектов некоторого класса на себя. Диаграмма на рис. 12.35 моделирует неориентированный граф при помощи рекурсивной ассоциации.

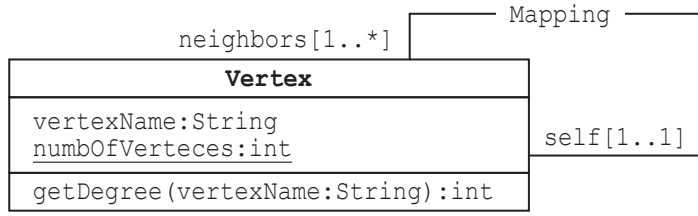



**Рис. 12.35.** Модель неориентированного графа с использованием рекурсивной ассоциации

Графический символ ассоциации не снабжен стрелками, и это означает, что навигация между объектами класса `Vertex` не определена. Атрибутами класса вершин являются: имя вершины (`vertexName`) и количество вершин (`numbOfVerteces`). Поведение описано методом-запросом (`getDegree`), который для каждой вершины, заданной именем, возвращает ее степень (количество инцидентных ребер). Имя ассоциации — `Mapping` (отображение). Один из полюсов ассоциации имеет имя `self` и соответствует текущему объекту класса `Vertex`. Второй полюс имеет имя `neighbors` (соседи) и моделирует множество узлов, непосредственно связанных с данным узлом. Его множественность описана выражением `[1..*]`, которое означает, что произвольный узел может быть связан с одним или некоторым количеством узлов.

На рис. 12.36 приведено отображение модели на рис. 12.35 в программный код. Поскольку рекурсивная ассоциация, при помощи которой мы моделируем неориентированный граф, моделирует связи между объектами, имеющими одинаковый ролевой статус, то имеется только один вариант ее отображения в код.

```
class Vertex {
   // поля
   public String vertexName;
   public Vertex neighbors[]; // определено рекурсивно
   public static final int numbOfVerteces;
   // метод
   public int getDegree(String vertexName){
   // код метода getDegree
   }
}
```

**Рис. 12.36.** Код, соответствующий модели на рис. 12.35

При отображении модели, приведенной на рис. 12.35, в код использован только один из полюсов рекурсивной ассоциации с именем `neighbors`. Поле `neighbors` имеет тип `Vertex` и для каждого объекта класса `Vertex` содержит множество ссылок на объекты этого же класса `Vertex`.

Диаграмма на рис. 12.37 иллюстрирует случай, когда объекты, связанные рекурсивной ассоциацией, имеют различный ролевой статус.

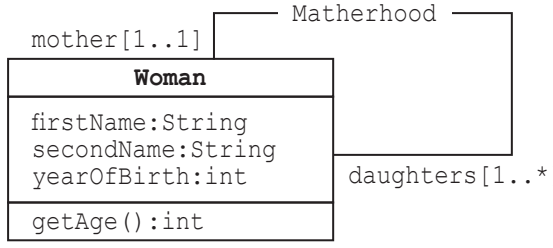



**Рис. 12.37.** Модель отношения «материнство» с использованием рекурсивной ассоциации

Диаграмма классов на рис. 12.37 моделирует отношение материнства. Отношение материнства рассматривается как сущность, связывающая между собой объекты класса `Woman` (женщина). Связи между объектами моделируются рекурсивной ассоциацией с именем `Matherhood` (материнство). Объекты имеют два различных ролевых статуса, отраженных в именах полюсов ассоциации. Одни их них моделируют матерей, и им соответствует полюс ассоциации с именем `mother` (мать), а другие — дочерей, и им соответствует полюс ассоциации с именем `daughters` (дочери). Множественность полюса с именем `mother` означает, что у дочери есть/была только одна мать, а множественность полюса именем `daughters` утверждает, что у матери есть/были одна или несколько дочерей.

Класс `Woman` характеризуется тремя атрибутами: имя (`firstName`), фамилия (`secondName`) и год рождения (`yearOfBirth`). Поведение класса `Woman` описано методом-запросом с именем `getAge`, возвращающим возраст.

На рис. 12.38 приведены два варианта отображения модели на рис. 12.37 в программный код. Коды отличаются тем, какой из двух полюсов рекурсивной ассоциации учитывается при отображении модели.

```
class Woman {
   // поля
   public String firstName;
   public String secondName;
   public int yearOfBirth;
   public Woman daughters[]; // определено рекурсивно
   // метод
   public int getAge(){
   // код метода getAge
   }
}
```

а)

```
class Woman {
   // поля
   public String firstName;
   public String secondName;
   public int yearOfBirth;
   public Woman mother;     // определено рекурсивно
   // метод
   public int getAge(){
   // код метода getAge
   }
}
```

б)

**Рис. 12.38.** Коды, соответствующие модели на рис. 12.37: а) учитывается полюс `daughters`; б) учитывается полюс `mother`

В варианте а) при отображении модели в код учитывался полюс с именем `daughters`. Поэтому в список полей класса `Woman` включено рекурсивно объявленное поле `daughters`. Поле хранит ссылки на объекты класса `Woman`, которые моделируют дочерей.

В варианте б) при отображении модели в код учитывался полюс с именем `mother`. Поэтому в список полей класса `Woman` включено рекурсивно объявленное поле `mother`, хранящее ссылку на объект класса `Woman`, который моделирует мать.

### 12.5.7. Ограничения для отношения типа ассоциация

Ограничения для отношения типа ассоциация целесообразно рассматривать, разделив их на две группы: (1) ограничения для отношения типа ассоциация-связь и (2) ограничения для отношения типа ассоциация-класс.

Полюса ассоциации-связи отображаются в поля, определяемые ассоциацией, и поэтому единственные OCL-ограничения, применимые для этих ассоциаций, — это ограничения, специфицирующие начальные значения полей, либо ограничения, специфицирующие правила формирования производных полей (в том случае, если полюс отображается в производное поле). Поэтому для ограничения типа ассоциация-связь целесообразно использовать естественно-языковую форму записи ограничений, которая не стесняет разработчика модели, и позволяет включать в модель самые различные ограничения, а не только ограничения полюсов, как в случае использования OCL. Высокая степень неопределённости такой формы записи ограничений является платой за универсальность и гибкость.

Рис. 12.39 иллюстрирует использование естественно-языковой формы записи ограничений для случая ассоциации-связи. Диаграмма на рис. 12.39 моделирует новые сущности, образуемые ассоциацией одного из объектов класса `Screen` (экран) и некоторого количества объектов класса `Window` (окно). Полюс, примыкающий к классу `Window`, помечен ограничением `{ordered}` (упорядочены). Это ограничение означает, что окна расположены на экране не произвольно, а в некотором порядке. Например, окна располагаются на экране в порядке их появления с перекрытием, а полностью видимым является последнее окно. Отметим, что хотя ограничение и записано после имени и множественности полюса, оно характеризует именно ассоциацию, а не объекты класса `Window`.

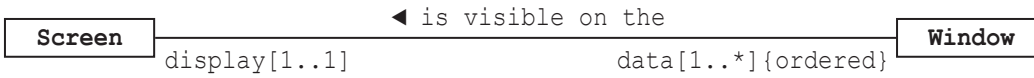


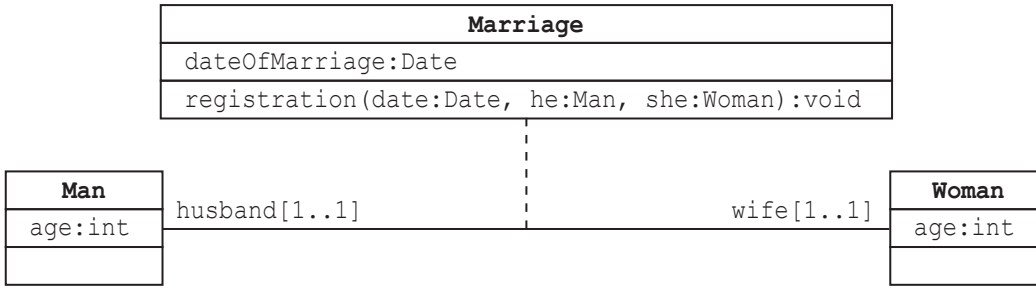

**Рис. 12.39.** Пример естественно-языковой формы записи ограничений для отношения типа ассоциация-связь

Ассоциации-классы позволяют применить весь арсенал OCL-ограничений с целью детального и определённого уточнения отношения типа ассоциация.

Класс, моделирующий ассоциацию, может быть ограничен при помощи OCL-ограничений, специфицирующих его инварианты. Напомним, что инвариант класса — это булево выражение, которое должно оставаться истинным на протяжении существования любого объекта этого класса. Если OCL-ограничение специфицирует инвариант ассоциации-класса, то оно определяет некоторое неизменное свойство отношения, объединяющего объекты в новую сущность. Например, отношение типа ассоциация с именем `Marriage` (супружество), установленное между классами `Man` (мужчина) и `Woman` (женщина) и представленное в виде класса, может быть уточнено инвариантом, ограничивающим возраст мужчин и женщин, вступающих в брак. В общем случае, к ассоциации-классу применимы все OCL-ограничения, предназначенные для ограничения класса.

Поля ассоциации-класса могут быть уточнены OCL-ограничениями, специфицирующими их начальные значения, а если среди полей имеются производные, то и ограничениями, специфицирующими правила формирования производных полей.

Методы ассоциации-класса могут быть уточнены OCL-ограничениями, специфицирующими их предусловия и постусловия а, в общем случае, всеми ограничениями, предназначенными для ограничения методов. Диаграмма на рис. 12.40 иллюстрирует использование OCL-ограничений для уточнения ассоциации-класса.




```
context Marriage::registration(date:Date, he:Man, she:Woman):void
pre: husband.age > 18 and wife.age > 18
post: true
```

**Рис. 12.40.** Пример использования OCL-ограничений для уточнения ассоциации-класса

Диаграмма на рис. 12.40 моделирует структуру системы, состоящей из двух ассоциированных классов `Man` и `Woman`. Ассоциация, установленная между классами `Man` и `Woman`, имеет имя `Marriage` и представлена в виде класса. Представление ассоциации в виде класса позволяет описать ассоциацию при помощи поля `dateOfMarriage` (дата заключения брака) и метода `registration` (регистрация брака). Метод `registration` уточнён при помощи OCL-ограничения, специфицирующего предусловие выполнения этого метода. Напомним, что предусловие должно быть истинным в тот момент, когда метод начинает выполняться. Если предусловие принимает ложное значение, то метод не выполняется. Предусловие метода `registration` специфицирует

возрастные ограничения на вступление в брак и запрещает регистрировать брак в том случае, когда возраст мужчины либо женщины меньше восемнадцати лет.

В OCL-выражении предусловия метода `registration` присутствуют поля классов `Man` и `Woman`. Поскольку метод `registration` объявлен в другом классе (классе `Marriage`), то при записи этих полей использованы составные имена: `husbant.age` и `wife.age`, отражающие навигацию от класса `Marriage` к классам `Man` и `Woman`. OCL позволяет при записи OCL-выражений использовать поля нескольких ассоциированных классов. Из диаграммы на рис. 12.40 явно не следует, что класс `Marriage` ассоциирован с классами `Man` и `Woman`. Однако, ранее (см. рис. 12.33), мы показали, что такие ассоциации имеют место.

### 12.5.8. Учет навигации в OCL-выражениях

При записи OCL-ограничений навигация используется во всех случаях, когда в OCL-выражении используются не только поля класса, указанного в контексте, но и поля «чужих» классов, ассоциированных с классом, указанном в контексте. При этом «чужие» классы могут быть ассоциированы с классом контекста как непосредственно, так и опосредованно через один или несколько классов-посредников.

Навигация от класса, указанного в контексте, к непосредственно и опосредованно ассоциированному классу в OCL-выражении отображается при помощи составного имени, описывающего «путь» от класса, указанного в контексте, к полю ассоциированного класса. В качестве элементов составного имени используются имена полюсов ассоциаций.

На рис. 12.41 приведена диаграмма классов, которая будет использована для иллюстрации структуры составного имени, необходимого для осуществления навигации в системе ассоциированных классов.

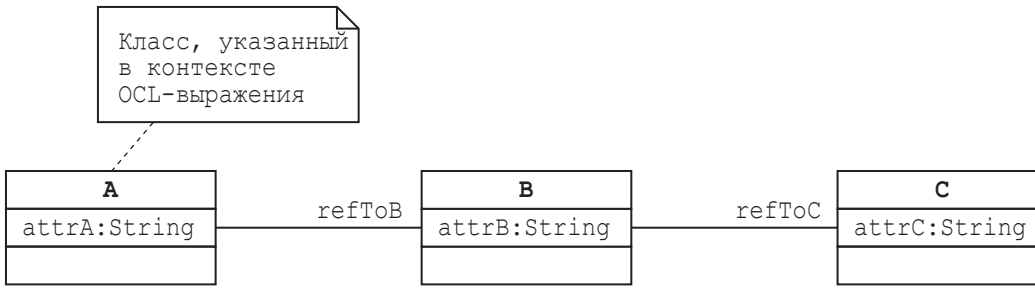



**Рис. 12.41.** Навигация между ассоциированными классами при помощи составных имен

Модель, приведенная на рис. 12.41, включает три ассоциированных класса с именами `A`, `B` и `C`.

Каждый из классов содержит по одному полю, определенному в символе класса. Это поля с именами `attrA`, `attrB` и `attrC`. Кроме этих полей, классы `A` и `B` содержат

по одному полю, определяемому полюсом ассоциации. Это поле с именем `refToB` для класса `A` и поле с именем `refToC` для класса `B`.

Пусть нам необходимо составить OCL-выражение для одного из ограничений класса `A`. Если в этом OCL-выражении необходимо использовать поле `attrB` класса `B`, то имя поля `attrB` должно быть составными и иметь структуру, приведенную на рис. 12.42.

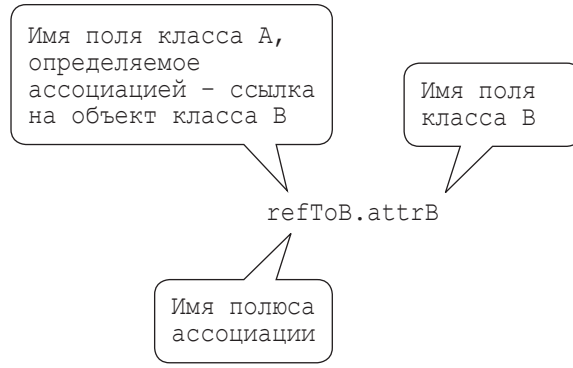



**Рис. 12.42.** Структура составного имени в случае навигации между двумя непосредственно ассоциированными классами `A` и `B` на рис. 12.41

Рассмотрим, теперь, случай, когда в OCL-выражение, в ограничении класса `A` необходимо использовать поле `attrC` класса `C`. Класс `C` ассоциирован с классом `A` опосредованно через класс `B` . На рис. 12.43 приведена структура составного имени, которое должно использоваться в этом случае.

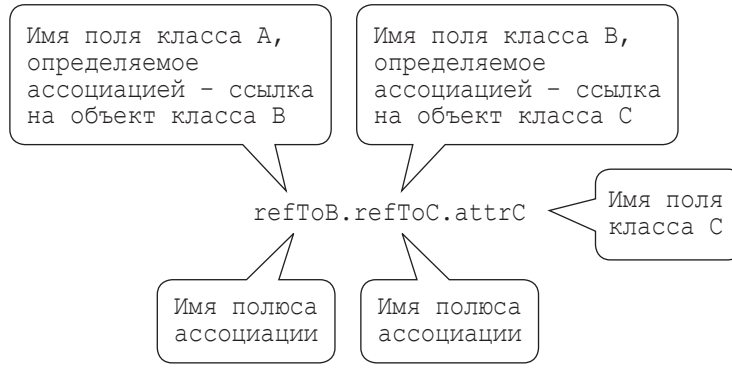



**Рис. 12.43.** Структура составного имени в случае опосредованной навигации в системе из трех ассоциированных, классов на рис. 4.41

Описанный способ навигации между ассоциированными классами с помощью составных имен использовались нами ранее при записи OCL-выражений в системе учета продажи билетов на авиарейсы (рис. 11.45) и в примере использования OCL-ограничения для уточнения ассоциации-класса (рис. 12.40).

## 12.6. Отношения типа часть-целое

Язык UML позволяет моделировать отношение между структурным элементом программы, рассматриваемым как целое, и его составными частями. Для этой цели UML предоставляет два вида отношений, которые носят наименование *композиция* и *агрегация*. Отношения типа композиция и агрегация можно рассматривать как частные случаи отношения типа ассоциация. Они моделируют систему, в которой новая сущность-целое образуется путем ассоциации ее частей.

Отношение типа композиция используется в тех случаях, когда целое и его части характеризуются следующими признаками.

- Часть в составе целого существует до тех пор, пока существует целое, и после уничтожения целого автоматически уничтожается. Этот признак называется признаком *каскадного удаления*.
- Часть принадлежит только одному целому, и в один и тот же момент не может принадлежать нескольким целым.
- Части, из которых состоит целое, обладают различной структурой.

Если мы рассматриваем деревянный дом как целое, а его крышу, стены, окна, двери и т. п. — как его части, то для моделирования отношения между домом и его частями необходимо использовать отношение типа композиция. Действительно, крыша дома (часть в составе целого) не существует вне дома (целое), а если дом уничтожается пожаром, то автоматически уничтожается его крыша, стены, окна, двери и т. п. Конкретная часть дома, например, входная дверь, входит в состав только этого дома и не может одновременно входить в состав нескольких домов. Части дома различны, с точки зрения их структуры. Крыша и окно имеют различную структуру.

Отношение типа агрегация используется в тех случаях, когда целое и его части характеризуются признаками.

- Части целого могут существовать вне целого и после уничтожения целого продолжают своё существование.
- Часть в один и тот же момент может входить в состав нескольких целых.
- Части, из которых состоит целое, обладают одинаковой структурой.

Если мы рассматриваем Общество любителей научной фантастики как целое, а его членов — как части этого целого, то для моделирования отношения между Обществом любителей научной фантастики и его членами необходимо использовать отношение типа агрегация. Действительно, члены Общества любителей научной фантастики (части целого) могут существовать вне Общества (целое), а если Общество перестаёт существовать, то его члены продолжают своё существование. Член Общества любителей научной фантастики может в то же время быть членом другой общественной организации, например, Клуба рыболовов-любителей. И, наконец, все члены Общества любителей научной фантастики — люди и, следовательно, обладают одинаковой структурой.

### 12.6.1. Отношение типа композиция

При использовании отношения типа *композиция* принята следующая терминология. Класс, который рассматривается как целое, называется *композитом*, а класс, который рассматривается как часть целого, называется *компонентом*. С использование этих терминов характерные признаки отношения типа композиция могут быть сформулированы следующим образом.

- Компоненты в составе композита существуют до тех пор, пока существует композит, и после удаления композита автоматически удаляются. Иными словами, имеет место каскадное удаление компонентов после удаления композита.
- Компонент принадлежит только одному композиту и в один и тот же момент времени не может принадлежать нескольким композитам.
- Компоненты, из которых состоит композит, обладают различной структурой.

На рис. 12.44 приведен пример, иллюстрирующий использование отношения типа композиция на диаграмме классов.

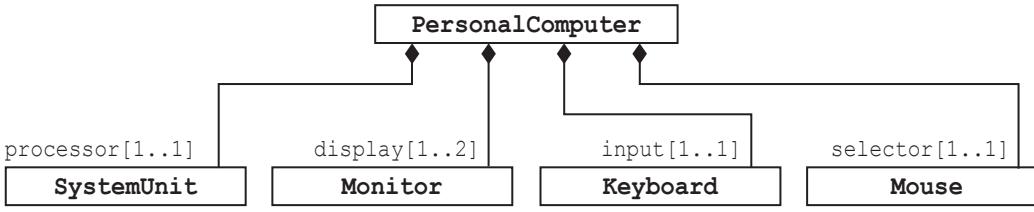



**Рис. 12.44.** Пример диаграммы классов с отношением типа композиции

Графическим символом отношения типа композиция является отрезок прямой, один из концов которой снабжён зачернённым ромбом. Ромб указывает на класс, являющийся композитом. Пример, приведенный на рис. 12.44, моделирует упрощённое представление об основных компонентах персонального компьютера.

Диаграмма на рис. 12.44 показывает, что класс `PersonalComputer` (персональный компьютер) является композитом четырёх компонентов: `SystemUnit` (системный блок), `Monitor` (монитор), `Keyboard` (клавиатура) и `Mouse` (манипулятор типа «мышь»).

Поскольку отношение типа композиция является частным случаем отношения типа ассоциация, то к нему применимы все характеристики ассоциации: имя ассоциации, имена и множественности полюсов, а также направление навигации.

Первой отличительной особенностью отношения типа композиция как частного случая ассоциации является одинаковость имени композиции для всех случаев применения этого отношения. Имя любого отношения типа композиция всегда одно и то же и может быть записано в виде: «`consists of`» (состоит из). Поэтому на диаграмме классов имя композиции, как правило, не указывают.

Второй отличительной особенностью отношения типа композиция является постоянная множественность полюса, примыкающего к композиту, имеющая значение

[1..1]. Поэтому на диаграмме классов имя и множественность полюса, примыкающего к композиту, как правило, не указывают. Множественность полюса на стороне компонентов различна, поэтому на диаграмме классов необходимо указывать имя и множественность полюсов для компонентов. Полюса компонентов на рис. 12.44 означают следующее.

Полюс processor[1..1] (процессор) означает, что роль объекта класса SystemUnit в композиции заключается в обработке информации и что в состав одного объекта класса PersonalComputer входит один объект класса SystemUnit.

Полюс display[1..2] (дисплей) означает, что роль объекта класса Monitor в композиции заключается в отображении информации и что один объект класса PersonalComputer может включать один или два объекта класса Monitor.

Полюс input[1..1] (ввод) означает, что роль объекта класса Keyboard в композиции — ввод информации и что в состав одного объекта класса PersonalComputer входит один объект класса Keyboard.

Полюс selector[1..1] (селектор) означает, что роль объекта класса Mouse в композиции заключается в выборе объектов, отображаемых на экране монитора, и что один объект класса PersonalComputer состоит из одного объекта класса Mouse.

Направление навигации для отношения типа композиция можно указывать несколькими способами. Универсальным способом является дополнительная стрелка, размещённая над графическим символом отношения типа композиция. Этот способ является единственно возможным, если мы указываем однонаправленную навигацию от компонентов к композиту. Однонаправленная навигация от композита к компонентам может быть указана так, как это показано на рис. 12.45.

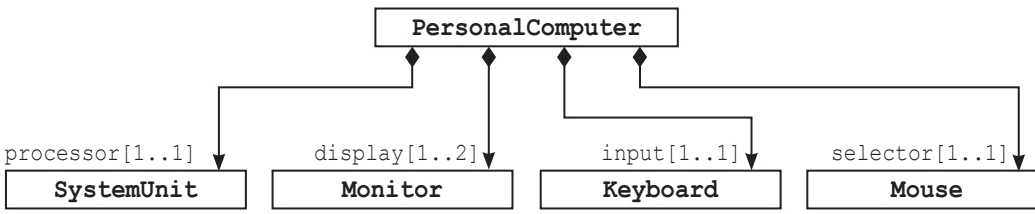



**Рис. 12.45.** Моделирование направления навигации от композита к компонентам

Направление навигации на рис. 12.45 означает, во-первых, что каждый объект класса PersonalComputer «знает», с какими объектами из классов SystemUnit, Monitor, Keyboard и Mouse он связан, а во-вторых, что ни один из объектов перечисленных классов «не знает», в состав какого объекта класса PersonalComputer он входит.

На рис. 12.46 приведен пример диаграммы классов, которая иллюстрирует совместное использование отношений типа обобщение-специализация и композиция для моделирования структуры чертежа.

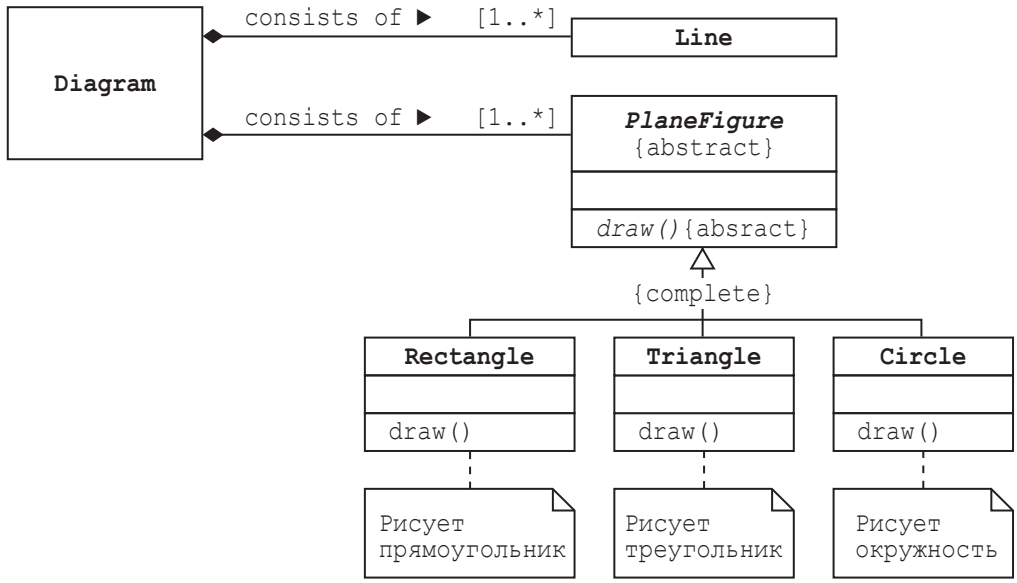



**Рис. 12.46.** Диаграмма классов, иллюстрирующая совместное использование отношений типа обобщение-специализация и композиция

Диаграмма на рис. 12.46 моделирует структуру простого чертежа. Простые чертежи, представленные классом `Diagram`, состоят из плоских фигур (объекты класса `PlaneFigure`) и линий (объекты класса `Line`). Любой простой чертёж состоит хотя бы из одной плоской фигуры и одной линии. Отношение «часть-целое» между классом `Diagram` (целое) и классами `PlaneFigure` и `Line` (части) моделируется отношением типа композиция. Класс плоских фигур (`PlaneFigure`) разделяется на три подкласса: `Rectangle` (прямоугольник); `Triangle` (треугольник) и `Circle` (окружность). Модель предполагает, что множество подклассов полное. Это отмечено ограничением декомпозиции `complete`. Полнота множества подклассов позволила представить класс `PlaneFigure` как абстрактный. Подклассы класса `PlaneFigure` наследуют все его члены и реализуют полиморфный метод `draw` в соответствии с типом плоской фигуры.

### 12.6.2. Отношение типа агрегация

При использовании отношения типа *агрегация* принята следующая терминология. Класс, который рассматривается как целое, называется *агрегатом*, а класс, который рассматривается как часть целого — *конституентом*.

С использование этих терминов характерные признаки отношения типа агрегация могут быть сформулированы следующим образом.

- Конституенты существуют самостоятельно и после уничтожения агрегата продолжают своё существование.
- Конституент в один и тот же момент времени может входить в состав нескольких агрегатов.
- Конституенты обладают одинаковой структурой. Поскольку одинаковой структурой обладают объекты одного и того же класса, то этот признак означает, что *конституентом может быть только один класс*. С учётом того, что агрегация является частным случаем ассоциации, можно утверждать, что *только бинарная ассоциация может быть агрегацией*.

На рис. 12.47 приведен пример диаграммы классов, в которой использовано отношение типа агрегация. Диаграмма моделирует упрощённую структуру официальной документации UML и показывает, что документация, представленная классом `UMLSpecification` (спецификация UML) состоит из отдельных разделов, представленных классом `Section` (раздел).

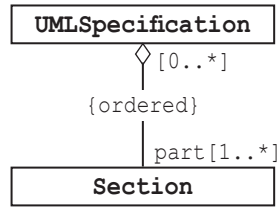



**Рис. 12.47.** Пример диаграммы классов с отношением типа агрегация

Графический символ отношения типа агрегация аналогичен графическому символу отношения типа композиция. Отличие заключается в том, что ромб, указывающий на агрегат, не зачерняется.

Поскольку объект- конституент может быть частью более, чем одного объекта-агрегата, то множественность полюса на стороне агрегата может быть произвольной. Поэтому оба полюса, как на стороне конституента, так и на стороне агрегата снабжаются именем и множественностью.

В примере на рис. 12.47 на стороне агрегата (класс `UMLSpecification`) указана множественность полюса в виде выражения `[0..*]`. Это означает, что некоторый раздел (объект класса `Section`) может входить в состав или нескольких официальных документаций, или не входить в состав ни одной документации. Полюс на стороне конституента (класс `Section`) описан именем `part` (часть) и множественностью в виде выражения `[1..*]`. Это означает, что одна документация (объект класса `UMLSpecification`) может включать один или несколько разделов.

Ограничение `{ordered}` *уточняет отношение типа агрегация* и указывает на то, что имеется некоторый порядок вхождения разделов в документацию (например, разделы располагаются в документации в порядке возрастания их номеров).

### 12.6.3. Отображение отношений типа композиция и агрегация в программный код

Отображение отношений типа композиция и агрегация в программный код осуществляется таким же образом, как и отображение отношения типа ассоциация-связь. В объявления классов, моделирующих целое и его части, включаются поля, хранящие ссылки, обеспечивающие направление навигации, указанное на диаграмме. Имена этих полей формируются из имён соответствующих полюсов, а типы задаются именами классов, моделирующих целое и его части.

Проблемой кодирования отношения типа композиция может быть обеспечение каскадного удаления объектов-компонентов при удалении объекта-композита. Однако при использовании языка программирования Java эта проблема решается автоматически программой «сборщик мусора». Как будет показано ниже, ссылки на объекты-компоненты включаются в атрибутивную модель объекта-композита. Поэтому, если удаляется объект-композит, то удаляются ссылки на его объекты-компоненты, и, следовательно, сами объекты-компоненты автоматически удаляются программой «сборщик мусора».

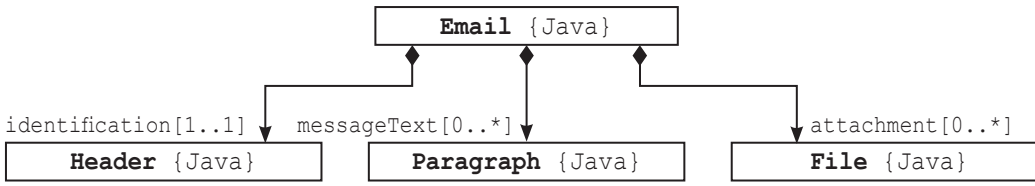
Таким образом, при отображении отношений типа композиция и агрегация в программный код не учитываются специфические особенности этих отношений, рассмотренные выше. На уровне кода как композиция, так и агрегация неотличимы от ассоциации-связи. Однако на уровне UML различие между отношениями типа композиция и агрегация позволяет создавать более точные и адекватные модели.

На рис. 12.48 приведен пример отображения отношения типа композиция в программный код. Диаграмма классов, приведенная на рис. 12.48, моделирует структуру электронного письма и утверждает, что класс `Email` (электронное письмо) является композитом и состоит из следующих компонентов: класс `Header` (заголовок), класс `Paragraph` (параграф) и класс `File` (файл).

Объекты класса `Header` играют роль идентификаторов электронного письма (имя полюса `identification`). В состав одного электронного письма входит один заголовок, поэтому множественность полюса описана выражением `[1..1]`. Объекты класса `Paragraph` играют роль текста сообщения (имя полюса `messageText`). Одно электронное письмо может либо не содержать ни одного текстового параграфа (пустое письмо), либо содержать несколько текстовых параграфов. Множественность полюса описана выражением `[0..*]`. Объекты класса `File` играют роль приложения к письму (имя полюса `attachment`). Одно электронное письмо может либо не содержать ни одного приложения, либо содержать несколько приложений. Множественность полюса описана выражением `[0..*]`.

В диаграмме классов на рис. 12.48 отношение между классом, моделирующим электронное письмо, рассматриваемым как целое, и его частями моделируется отношением типа композиция с однонаправленной навигацией от композита к компонентам. Это означает, что располагая экземпляром электронного письма, можно

получить все связанные с ним компоненты: заголовок, текстовые сообщения и приложения. Однако обратная навигация, от компонентов письма к самому письму невозможна. Поэтому поля, хранящие ссылки, обеспечивающие требуемую навигацию, размещаются в классе `Email` и *являются частью его атрибутивной модели*. Этот факт позволяет рассматривать отношение типа композиция с навигацией от композита к компонентам как *способ графического представления атрибутивной модели композита*.




```
public class Email
{
...
   private Header identification;
   private Paragraph messageText[];
   private File attachment[];
...
}
```

**Рис. 12.48.** Отображение отношения типа композиция в программный код. Случай однонаправленной навигации от композита к компонентам

## 12.7. Отношение типа зависимость

Отношение типа *зависимость*, в общем случае, применяется для моделирования случая, когда функционирование одного или нескольких элементов системы зависит от других элементов этой же системы. Отношение типа зависимость называется также отношением типа *снабженец-клиент*, в котором клиент зависит от снабженца. Применительно к диаграмме классов, зависимость класса-клиента от класса-снабженца понимается в том смысле, что изменения в классе-снабженце приводят к необходимости выполнения изменений в классе-клиенте. Поэтому специфицирование класса-клиента будет неполным без класса-снабженца.

Графическим символом отношения типа зависимость является пунктирная линия со стрелкой, указывающей на независимый класс (класс-снабженец). Рис. 12.49 иллюстрирует изображение отношения типа зависимость на диаграмме классов.

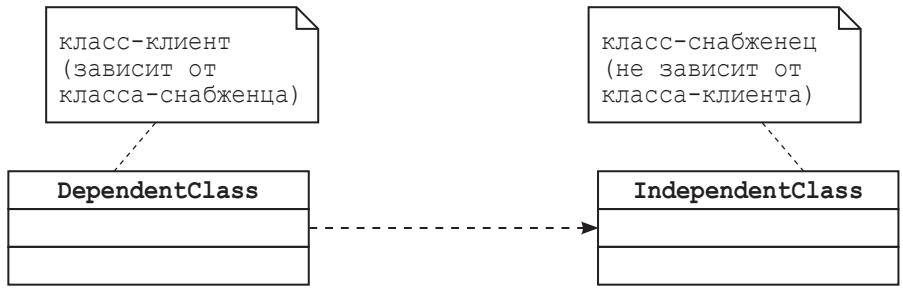



**Рис. 12.49.** Изображение отношения типа зависимость на диаграмме классов

На рис 12.50 приведен пример, иллюстрирующий использование отношения типа зависимость на диаграмме классов.

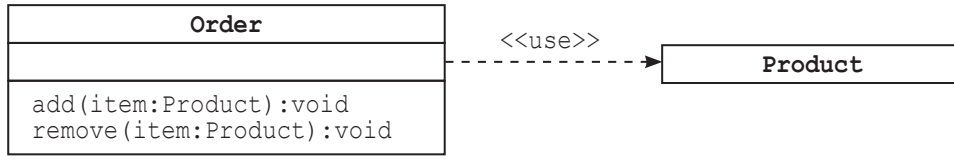



**Рис. 12.50.** Пример диаграммы классов с отношением типа зависимость

В примере, приведенном на рис. 12.50, зависимый класс `Order` (заказ) содержит ссылки на объекты независимого класса `Product` (товар) в виде входных параметров методов `add` (добавить) и `remove` (удалить). Оба метода используют входной параметр с именем `item` (пункт) типа `Product`. Класс `Product` снабжает класс `Order` типом `Product` и находится с классом `Order` в отношении типа зависимость. Ясно, что любые изменения в классе `Product` должны быть учтены в классе `Order`.

Ранее мы отметили, что в отношении типа зависимость класс-снабженец снабжает типом класс-клиент. Это требует некоторых пояснений. Важен вопрос о том, каким образом тип, поставляемый классу-клиенту классом-снабженцем, используется в классе-клиенте. Ранее, при изучении отношения типа ассоциация, мы показали, что если тип внешнего класса используется в атрибутивной модели данного класса, то отношение между данным классом и внешним классом моделируется ассоциацией или композицией, а не зависимостью. Отсюда следует, что *тип, которым класс-снабженец снабжает класс-клиент, не может использоваться для специфицирования полей класса-клиента*, поскольку такой случай моделируется отношением типа ассоциация. Тип, который класс-клиент получает от класса-снабженца, используется классом-клиентом для специфицирования методов в качестве либо (1) типа одного из параметров метода; либо (2) типа объекта, создаваемого методом при помощи предложения со служебным словом `new`.

Отношение типа зависимость может быть помечено стереотипом для уточнения его семантики. Практически во всех случаях отношение типа зависимость можно снабдить предопределённым стереотипом <<use>> (использует). Включение этого стереотипа в модель почти ничего не добавляет в понимание отношения типа зависимость, поскольку выражает его сущность. По этой причине стереотип <<use>> применяется редко.

В таблице на рис. 12.51 приведен список некоторых предопределённых стереотипов, которые можно использовать для уточнения отношения типа зависимость.

С целью уточнения семантики отношения типа зависимость программист может использовать не только предопределённые стереотипы, но и вводить свои собственные.

| Имя стереотипа | Назначение стереотипа |
|---|---|
| <<call>> | Метод класса-клиента может вызывать метод класса-снабженца. |
| <<create>><br><<instantiate>> | Метод класса-клиента может создавать объект класса-снабженца. |
| <<send>> | Метод класса-клиента посылает сигнал классу-снабженцу. |
| <<use>> | Методы класса-клиента каким-то образом используют тип, задаваемый классом-снабженцем. |

**Рис. 12.51.** Стандартные стереотипы, используемые для уточнения отношения типа зависимость

## 12.8. Отношение типа реализация

Отношение типа *реализация* может рассматриваться как частный случай отношения типа зависимость в том случае, когда класс-клиент *реализует* спецификации класса-снабженца. В контексте языка программирования Java, отношение типа реализация удобно использовать для моделирования отношения между интерфейсом и классом, который его реализует. На рис. 12.52 приведена диаграмма классов, иллюстрирующая использование отношения типа реализация. Эта диаграмма моделирует класс с именем DialogueAgent (диалоговый агент), который реализует два интерфейса с именами Sound (звук) и Text (текст), но фокусирует внимание на том, что класс DialogueAgent *реализует* интерфейсы Sound и Text.

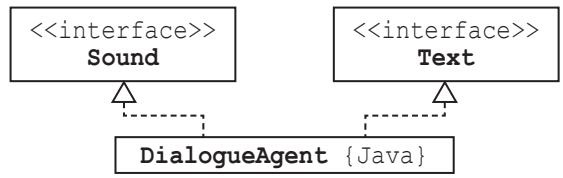



**Рис. 12.52.** Пример диаграммы классов с отношением типа реализация

Графическим символом отношения типа реализация является пунктирная линия с полым треугольником на конце. Треугольник указывает на класс-снабженец. Модель на рис. 12.52 показывает, что язык программирования Java разрешает одному классу реализовывать несколько интерфейсов. Класс `DialogueAgent` реализует два интерфейса с именами `Sound` и `Text`.

На рис. 12.53 приведена ещё одна модель с использовнием отношения типа реализация.

Модель, приведенная на рис. 12.53, показывает, что: (1) в языке программирования Java при объявлении класса можно использовать интерфейсы; а также, что (2) на уровне интерфейсов допустимо множественное наследование.

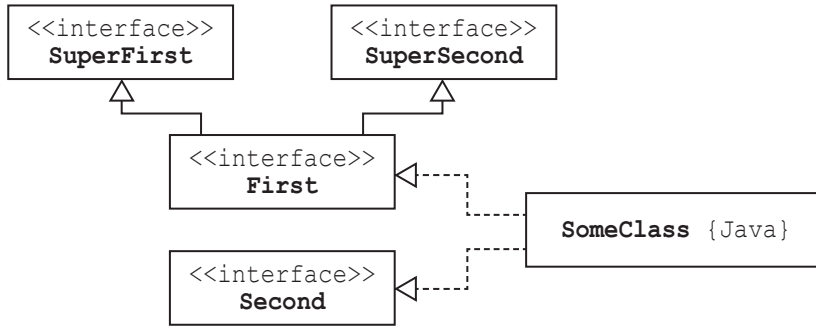



**Рис. 12.53.** Диаграмма классов, иллюстрирующая возможность реализации нескольких интерфейсов в одном классе и множественное наследование интерфейсов

Класс с именем `SomeClass` (некоторый класс) реализует два интерфейса: `First` (первый) и `Second` (второй). Интерфейс `First` является подинтерфейсом/подтипом и наследует члены у двух суперинтерфейсов/супертипов: `SuperFirst` и `SuperSecond`.

## Упражнения для практических занятий

12.1. Разработайте диаграмму классов, целью которой является построение дерева, классифицирующего работников больницы. Используя отношение типа обобщение-специализация, постройте диаграмму классов, включающую следующие классы: Личность, Пациент, Врач, Медсестра, Санитарка, Хирург, Терапевт. Отношение обобщение-специализация снабдите ограничениями декомпозиции суперкласса.

12.2. Подклассами класса Врач могут быть классы Хирург и Терапевт. Сформируйте любой из этих подклассов двумя способами: (1) путем

расширения атрибутивной модели и/или поведения; (2) путем переопределения атрибутивной модели и/или поведения. Представьте результат решения упражнения в виде двух диаграмм классов.

12.3. Разработайте диаграмму классов, моделирующую структуру университетской библиотеки. Будем считать, что в библиотеке хранятся только учебники, справочники и методические указания. Клиентами библиотеки являются преподаватели и студенты, однако работники библиотеки имеют право пользоваться книгами библиотеки. Структура университетской библиотеки должна включать, как минимум, следующие классы: Книги, Учебники, Справочники, Методические Указания, Клиенты, Преподаватели, Студенты, Библиотекари.

12.4. Разработайте диаграмму классов, моделирующую структуру простого чертежа. Будем считать, что простой чертеж состоит из прямоугольников, окружностей и линий, которые не обязательно являются прямыми. Будем считать также, что прямоугольник состоит из отрезков. Используя отношения типа композиция и агрегация, постройте диаграмму классов, включающую следующие классы: Чертеж, Прямоугольник, Окружность, Линия и Отрезок. При изображении отношений типа композиция и агрегация следует использовать все известные вам описатели этих отношений.

12.5. Используя отношение типа ассоциация, представьте структуру ориентированного графа в виде диаграммы классов. Рассматривайте ориентированный граф как совокупность (1) множества вершин и (2) такого отображения множества вершин на себя, в котором учитывается направление связи между вершинами. Специфицируйте свойства и поведение графа при помощи полей и методов.

12.6. Запишите программный код для модели, полученной в результате решения упражнения 4.5.

12.7. Представьте структуру двудольного ориентированного графа в виде диаграммы классов. Специфицируйте атрибуты и поведение графа при помощи полей и методов.

12.8. Запишите программный код для модели, полученной в результате решения упражнения 4.7.

12.9. На рис. 4.23 приведена диаграмма классов, моделирующая отношение «отцовство», представленное в виде ассоциации-класса. Уточните это отношение при помощи ограничений в виде OCL-предложений.

12.10. Разработайте диаграмму классов, моделирующую структуру системы формирования заказа на приобретение товаров. Один заказ позволяет приобрести несколько товаров. Каждый товар отображается в заказе отдельной строкой. Для всего заказа важными являются дата оформления заказа, номер заказа и общая стоимость заказа. Для отдельного товара важными являются: наименование товара, стоимость и количество товара. Клиенты, оформляющие заказ, делятся на корпоративных и частных. Система должна включать следующие классы: Заказ, Пункт заказа, Клиент, Корпоративный Клиент и Частный Клиент.

12.11. Предложите пример тернарной ассоциации и специфицируйте ее при помощи полей и методов. Не используйте пример, приведенный в пособии.

12.12. Сформулируйте условия использования композиции для моделирования системы. Можно ли использовать композицию для того, чтобы показать, что собака является композитом роста, веса, цвета и даты рождения. Обоснуйте Ваш ответ.

12.13. Булка хлеба, нарезанная на куски, состоит из хлебных кусков. Является ли отношение между булкой и её кусками композицией или агрегацией? Обоснуйте ответ соответствующим анализом.

12.14. Разработайте диаграмму классов, моделирующую две категории заказчиков предприятия: внешние заказчики и внутренние заказчики. Внешние заказчики не являются подразделениями предприятия, а внутренние заказчики представляют собой его подразделения. Например, внутренним заказчиком мебельной фабрики может быть хозрасчетная столовая этой же мебельной фабрики.

12.15. Разработайте диаграмму классов, которая моделирует структуру системы, включающую книгу, её автора, издателя и продавца. Будем считать, что книга состоит из разделов, которые состоят из глав. Каждая из глав, в свою очередь, включает несколько параграфов и рисунков. Модель должна учитывать информацию об авторе издателе и продавце.

Раздел 13

# МОДЕЛИРОВАНИЕ ПРОСТРАНСТВЕННОЙ СТРУКТУРЫ ПРИ ПОМОЩИ ДИАГРАММЫ ПАКЕТОВ И ДИАГРАММЫ ОБЪЕКТОВ

Пакет представляет собой группу родственных классов. Принципы группировки классов в пакеты определяются разработчиком модели в соответствии с целью моделирования. Например, в один пакет могут быть собраны классы элементов графических пользовательских интерфейсов, а в другой — классы, поддерживающие графический интерфейс и использующие эти элементы. Пакеты являются структурными блоками программной системы и, следовательно, модель пространственной структуры системы может быть построена в виде множества пакетов и отношений между ними. Диаграмма, позволяющая представлять такую модель, называется диаграммой пакетов. Одной из основных причин группировки классов в пакет является желание разработчика программной системы уменьшить количество структурных элементов модели и строить её из более крупных блоков. Однако, существует ещё одна причина группировки классов в пакет. Эта причина обусловлена свойством пакета обеспечивать *независимое пространство имён* для классов, входящих в его состав. Это важная причина, поскольку *одноимённые* структурные элементы программной системы не могут быть размещены в одном пространстве имён.

## 13.1. Графические символы пакета

Существует несколько способов изображения пакета на диаграмме пакетов, которые приведены на рис. 13.1.

Графический символ пакета представляет собой стилизованное изображение папки, принятое в графическом интерфейсе операционной системе Windows и в ряде приложений.

Рис. 13.1, а) иллюстрирует простейшее изображение графического символа пакета. Внутри стилизованного изображения папки записывается имя пакета `hospitalWard` (больничная палата).

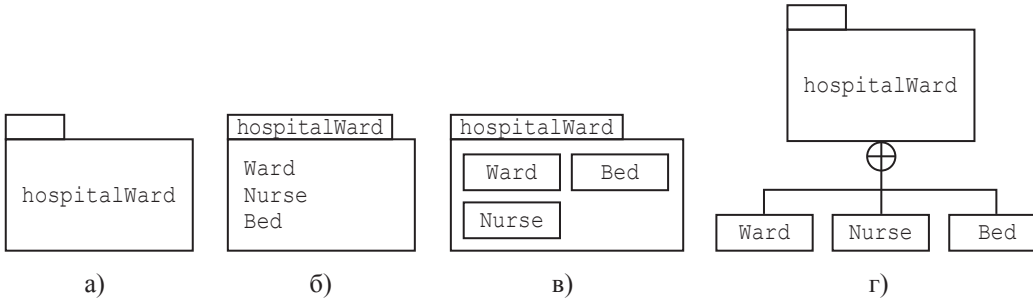



**Рис. 13.1.** Варианты изображения графического символа пакета

Рис. 13.1, б) и 13.1, в) иллюстрируют изображение графического символа пакета с указанием классов, включённых в пакет. Пакет с именем `hospitalWard` включает классы: `Ward` (палата), `Nurse` (медицинская сестра) и `Bed` (койка).

Диаграмма на рис. 13.1, г) семантически эквивалентна графическим символам пакета, приведенным на рис.13.1, б) и 13.1, в). В этой диаграмме явно указано, какие классы включены в пакет при помощи графического символа «якорь», который мы ранее использовали для моделирования вложенности классов.

В языке программирования Java для указания принадлежности класса некоторому пакету записывается предложение со служебным словом `package`, которое предшествует объявлению класса. Например,

```
package hospitalWard;
public class Ward {
   . . .
}
```

Пакет может содержать не только классы, но и другие пакеты. Уровень вложенности пакетов теоретически не ограничен. На рис. 13.2 приведены варианты графического символа пакета для случая вложенности пакетов. В левой части рис. 13.2 графический символ внутреннего пакета `hospitalWard` изображается внутри графического символа внешнего пакета `data`.


**Рис. 13.2.** Графическое изображение вложенности пакетов

В средней части рис. 13.2 для моделирования вложенности используется составное имя `data::hospitalWard`, в котором *вначале указывается имя внешнего пакета, затем имя вложенного пакета* и, если необходимо, имя класса. Разделителем является двойное двоеточие. Таким образом, составное имя

```
data::hospitalWard::Nurse
```

обозначает класс `Nurse`, находящийся в пакете `hospitalWard` , который, в свою очередь, вложен в пакет `data`.

В правой части рис. 13.2 вложенность моделируется явно, при помощи графического символа «якорь».

## 13.2. Доступ к классам, размещённым в пакетах

Пакет обеспечивает независимое пространство имён для своих классов. Это означает, что два класса с одним и тем же именем, которые не могут быть размещены в одном пакете, могут быть размещены в двух различных пакетах. Если класс помещён в пакет, то его полное имя должно включать имя пакета, в котором он находится. На рис. 13.3 приведены графические символы классов, имена которых отражают их принадлежность к различным пакетам.


**Рис. 13.3.** Полное имя класса, размещённого в пакете

В UML полное имя класса является составным и формируется описанным выше способом с использованием двойного двоеточия в качестве разделителя. В языке программирования Java в качестве разделителя используется точка. Таким образом, в Java-коде полные имена классов, приведенных на рис. 5.3, записываются в виде

`patient.Person` и `doctor.Person`

Классы, находящиеся в одном пакете, принадлежат одному пространству имён, поэтому они могут ссылаться друг на друга без указания полного имени. Классы, принадлежащие различным пакетам, должны использовать полное имя для взаимных ссылок. Проиллюстрируем это примером. На рис. 13.4 приведены графические символы пакетов `patient` и `hospitalWard`. Классы `Bed` и `Ward` находятся в одном и том же пакете `hospitalWard`. Поэтому атрибутивная модель класса `Bed`

может включать поле типа Ward без указания полного имени класса Ward. Однако, поскольку классы Bed и Person находятся в различных пакетах, то в коде на языке программирования Java для ссылки на класс Bed из класса Person необходимо указывать полное имя hospitalWard.Bed. В языке программирования Java это же правило распространяется на вложенные пакеты. Например, класс Bed, находящийся в пакете hospitalWard, должен использовать полное имя для доступа к классу, находящемуся в пакете data, несмотря на то, что пакет hospitalWard вложен в пакет data (см. рис. 13.2).

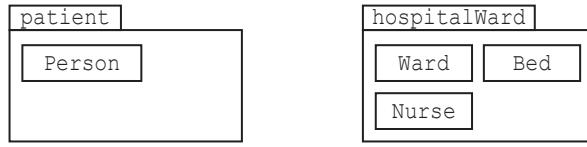



**Рис. 13.4.** Доступ к классам, размещённым в одном и том же и в различных пакетах

В последующих подразделах этого раздела мы узнаем о том, что доступ к классам некоторого пакета из другого пакета можно осуществить и без указания полного имени класса, если между пакетами установлено отношение импортирования классов.

Классы, помещённые в пакете, могут быть снабжены public (символ «+») или private (символ «–») префиксами видимости для указания их доступности из классов за пределами пакета. Класс, размещённый в некотором пакете и помеченный private префиксом видимости, доступен только для классов данного пакета и недоступен для классов других пакетов. Если класс помечен префиксом видимости public, то он доступен как для классов данного пакета, так и для классов других пакетов. На рис. 13.5 изображены те же пакеты, что и на рис. 13.4, но снабжённые префиксами видимости.

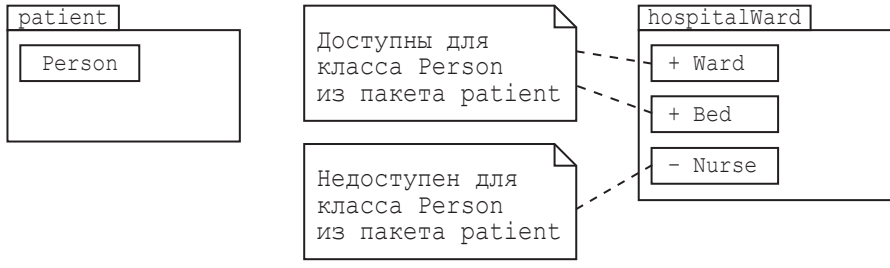



**Рис. 13.5.** Управление доступом к классам пакета при помощи префиксов видимости

Как видно на рис. 13.5, класс Nurse, размещённый в пакете hospitalWard, помечен private префиксом видимости и поэтому доступен только для классов пакета hospitalWard. Класс Person, находящийся в пакете patient, не имеет доступа к классу hospitalWard.Nurse.

В языке программирования Java префикс видимости класса, размещённого в пакете, записывается в виде соответствующего служебного слова в заголовке класса. Например,

```
package hospitalWard;
private class Nurse {
   . . .
}
```

Если при объявлении класса в его заголовке не указано служебное слово `public`, то, по умолчанию, это означает private уровень доступа, и, следовательно, класс доступен только для членов данного пакета и недоступен для классов за пределами пакета.

## 13.3. Отношение типа зависимость между пакетами и диаграмма пакетов

Как следует из предыдущего подраздела, в ряде случаев класс, размещённый в некотором пакете, должен использовать класс, размещённый в другом пакете. Для того, чтобы это было возможно, между пакетами должно быть установлено отношение типа зависимость.

Если класс пакета `packageA` использует класс пакета `packageB`, то пакет `packageA` зависит от пакета `packageB`. Диаграмма на рис. 13.6 иллюстрирует использование отношения типа зависимость на диаграмме пакетов.

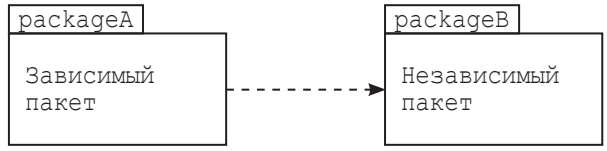



**Рис. 13.6.** Отношение типа зависимость между пакетами

Диаграмма пакетов представляет собой множество графических символов пакета и отношений между ними. В качестве отношения между пакетами чаще всего используется отношение типа зависимость. На рис. 13.7 приведен пример диаграммы пакетов, моделирующей структуру некоторого приложения для учёта больных в лечебном заведении.

Пакет `data` представляет собой объединение двух пакетов: `patient` и `hospitalWard`. Из диаграммы на рис. 13.7 следует, что пакет `patient` зависит от пакета `hospitalWard`. Эта зависимость может выражаться, например, в том, что класс `patient::Person` ссылается на класс `hospitalWard::Bed`.

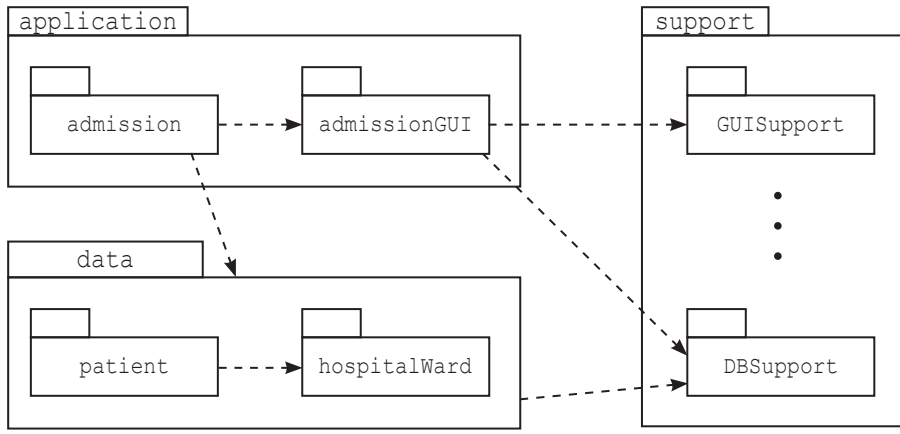



**Рис. 13.7.** Пример диаграммы пакетов

Пакет `support` (поддержка) объединяет пакеты и классы, приобретаемые с целью создания среды для функционирования приложения. Пакет `GUISupport` (поддержка графического пользовательского интерфейса (Graphical User Interface)) содержит классы для поддержки графического интерфейса приложения, а пакет `DBSupport` (поддержка базы данных (Data Base)) — классы для поддержки базы данных. Многоточие означает, что в пакете `support` имеются и другие элементы.

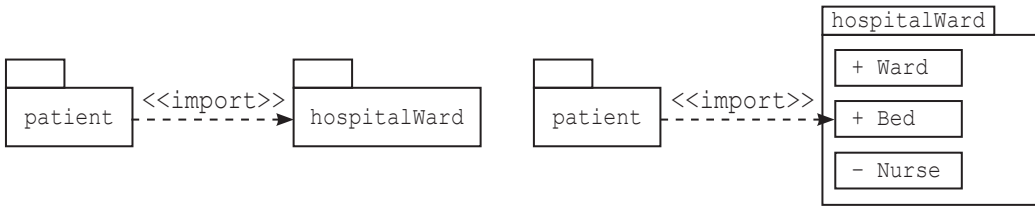
Пакет `application` (приложение) содержит два пакета. Пакет `admission` (приложения по приёму и выписке пациентов) содержит классы, которые реализуют приём и выписку пациентов. Поскольку классы этого пакета, очевидно, находятся в отношениях с классами, входящими в пакет `data`, то на диаграмме установлено отношение типа зависимость между пакетом `admission` и пакетом `data`. Отношение показывает, что пакет `admission` зависит от пакета `data`.

Пакет `admissionGUI` (графический пользовательский интерфейс по приёму и выписке пациентов) содержит классы, обеспечивающие графический пользовательский интерфейс той части приложения, которая обслуживает приём и выписку пациента. Классы этого пакета строятся с использованием классов пакета `GUISupport`. Поэтому между отмеченными классами установлено отношение типа зависимость.

## 13.4. Импортирование классов из пакета

Зависимость одного пакета от другого может иметь характер *импортирования классов*. Термин импортирование классов означает доступ к классам внешнего пакета без указания их полного имени. Когда между двумя пакетами установлено отношение импортирования классов, то это означает, что: (1) импорт осуществляется *из независимого пакета в зависимый* пакет; (2) классы зависимого пакета

попадают в пространство имён независимого пакета и, следовательно, могут обращаться к классам независимого пакета без указания их полного имени. Диаграммы на рис. 13.8 иллюстрируют использование отношения импортирования классов на диаграмме пакетов. Отношение импортирования классов не имеет специального графического символа. Для этой цели используется графический символ отношения типа зависимость, помеченный стереотипом <<import>>.




```
package patient
// импортирует все public классы
// пакета hospitalWard
import hospitalWard.*;
class Person {
   Ward hositalLocation;
{
```

```
package patient
// импортирует только класс Bed
import hospitalWard.Bed;
class Person {
   Bed wardLocation;
{
```

**Рис. 13.8.** Примеры использования отношения импортирования классов

Диаграмма пакетов в левой части рис. 13.8 иллюстрирует случай, когда пакет `patient` импортирует все классы пакета `hospitalWard`. Это означает, что классы пакета `patient` могут обращаться ко всем классам пакета `hospitalWard`, помеченным префиксом видимости `public`, без указания их полного имени. Диаграмма пакетов в правой части рис. 13.8 иллюстрирует случай, когда пакет `patient` импортирует только один класс с именем `Bed` из пакета `hospitalWard`. Это означает, что классы пакета `patient` могут обращаться без указания полного имени только к классу `Bed` пакета `hospitalWard`. Для доступа к остальным классам пакета `hospitalWard`, помеченных префиксом видимости `public`, необходимо указывать их полное имя. Импортирование классов из одного пакета в другой возможно только для классов, помеченных префиксом видимости `public`. Классы, помеченные `private` префиксом видимости, остаются невидимыми за пределами пакета и не подлежат импортированию. В примере на рис. 13.8 импортированию не подлежит класс `Nurse`.

В нижней части рис. 13.8 показано, каким образом отношение импортирования классов отображается в код на языке программирования Java. Для этой цели используется предложение со служебным словом `import`, после которого указывается полное имя класса, подлежащего импортированию. Например, `hospitalWard.Bed` (правая часть рис. 13.8). В том случае, когда импортированию подлежат все классы пакета, вместо имени импортируемого класса записывается символ «звёздочка». Например, `hospitalWard.*` (левая часть рис. 13.8).

## 13.5. Графические символы объекта

Структура класса в UML моделях определяется наборами его полей и методов. Когда объект создаётся при помощи класса, он получает структуру, подобную структуре этого класса. Отличия между структурой класса и структурой объекта этого класса заключаются в том, что: (1) объект включает только нестатические поля; (2) поля объекта содержат конкретные значения, определяющие состояние объекта для некоторого момента времени. Варианты графического символа объекта приведены на рис. 13.9.

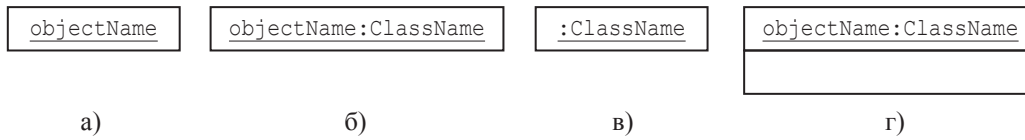



**Рис. 13.9.** Графические символы, используемые для изображения объекта

Графический символ объекта, в простейшем случае, представляет собой прямоугольник, в котором записывается имя объекта (первые три графических символа на рис. 13.9). Имя объекта записывается нежирным шрифтом и подчёркивается. Имя объекта может быть простым, составным или анонимным.

Простое имя объекта (вариант а) на рис. 13.9) представляет собой имя переменной-указателя, начинающееся с малой буквы. Как правило, это имя формируется при создании объекта при помощи предложения со служебным словом `new`.

Составное имя объекта (вариант б) на рис. 13.9) представляет собой имя объекта, за которым следует имя класса. В качестве разделителя используется символ двоеточие.

Анонимное имя объекта (вариант в) на рис. 13.9) представляет собой имя порождающего его класса с символом двоеточия впереди.

Графический символ объекта может включать отделение, в котором специфицируется состояние объекта (вариант г) на рис. 13.9). Состояние объекта специфицируется при помощи списка полей с указанием их значений. В список не обязательно включать все поля соответствующего класса, а лишь те, которые важны для модели. При специфицировании состояния объекта поля могут быть представлены либо только их именами, либо именами и типами. На рис. 13.10 приведены примеры вариантов графического символа объекта с отделением, содержащим поля и их значения.

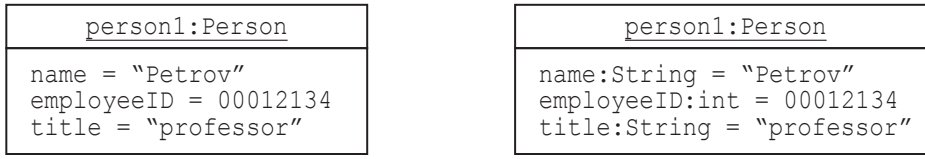



**Рис. 13.10.** Примеры графического символа объекта со списком значений полей

На рис. 13.10 приведены варианты графического символа объекта с именем `person1:Person`. Из имени объекта следует, что он создан при помощи класса `Person`, атрибутивная модель которого, кроме всего прочего, включает поля: `name` (фамилия), `employeeID` (идентификационный номер работника) и `title` (звание).

Поскольку значения полей изменяются в процессе функционирования программной системы, то состояние объекта соответствует некоторому моменту времени. Таким образом, графический символ объекта моделирует состояние этого объекта для некоторого момента времени по отношению к выбранному набору элементов атрибутивной модели. Отметим, что с течением времени изменяется не только состояние объектов, но и общее количество объектов программной системы.

## 13.6. Диаграмма объектов

Диаграмма классов, построенная на основе отношений типа *ассоциация, композиция и агрегация* может быть отображена в диаграмму объектов. *Диаграмма объектов* представляет собой «моментальный снимок» объектной системы для некоторого момента времени, и поэтому одна диаграмма классов может быть отображена во множество диаграмм объектов, в котором каждая из диаграмм соответствует конкретному моменту времени.

Диаграмма объектов моделирует состояние системы в некоторый момент времени и показывает, какие объекты находятся в памяти и каково значение их полей, а также, каким образом объекты связаны между собой.

Связи между объектами типизированы, а их типы соответствуют типам отношений на исходной диаграмме классов. Если отношение на диаграмме классов это — некоторая абстракция, имеющая условное обозначение в виде графического символа отношения, то связи на диаграмме объектов — это конкретная реализация отношения для заданного момента времени.

Диаграмма объектов представляет собой множество графических символов объектов и графических символов связей между объектами. В качестве графических символов связей используются графические символы ассоциации, композиции и агрегации. При трансляции диаграммы классов в диаграмму объектов целесообразно поля, которые на диаграмме классов задаются полюсами отношений типа ассоциация, композиция и агрегация, вводить в графический символ объекта. В противном случае они будут утеряны.

При помощи класса может быть создано сколь угодно много объектов. Однако, на диаграмме объектов возможно изобразить всего несколько объектов каждого из классов. Это одна из причин, по которой диаграмма объектов находит ограниченное применение. На рис. 13.11 приведена диаграмма классов (левая часть рис. 13.11) и один из возможных вариантов диаграммы объектов, соответствующей этой диаграмме классов (правая часть рис. 13.11).

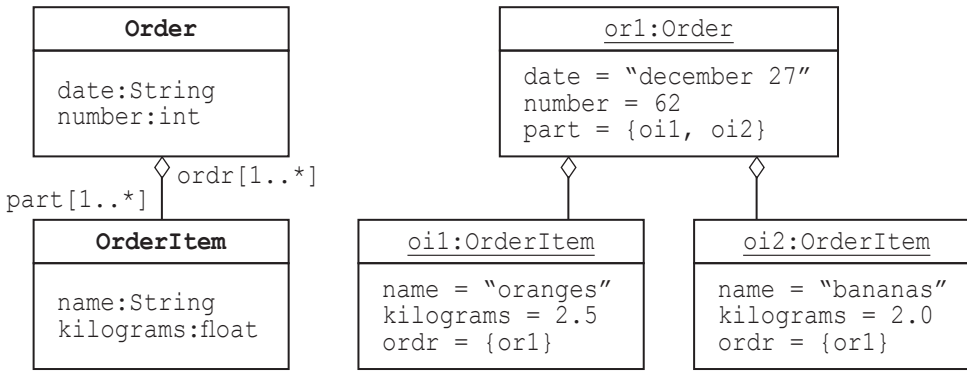



**Рис. 13.11.** Пример диаграммы классов с отношением типа агрегация и соответствующий вариант диаграммы объектов

В левой части рис. 13.11 приведена диаграмма классов, моделирующая структуру заказа. Класс `Order` (заказ) состоит из отдельных частей — пунктов заказа, моделируемых объектами класса `OrderItem`. Между классами `Order` и `OrderItem` установлено отношение типа агрегация. Из этого факта можно сделать некоторые заключения. Например, о том, что все строки заказа имеют одинаковую структуру. Множественности полюсов отношения типа агрегация означают, что: (1) один заказ состоит из одного или нескольких пунктов заказа (модель исключает пустые заказы) и что (2) один и тот же пункт может входить в состав нескольких заказов. Атрибутивная модель класса `Order` описана полями `date` (дата формирования заказа) и `number` (номер заказа), а атрибутивная модель класса `OrderItem` описана полями `name` (наименование товара) и `kilograms` (вес товара в килограммах).

В правой части рис. 13.11 приведен вариант диаграммы объектов, соответствующий диаграмме классов, изображённой в левой части. Диаграмма моделирует конкретный заказ, соответствующий некоторому моменту времени. Из диаграммы следует, что в данный момент времени заказ состоит из двух пунктов, при помощи которых заказываются апельсины и бананы. В диаграмме используется вариант графического символа объекта, который специфицирует состояние объекта. В списки полей этих объектов введены поля `part` и `ordr`, определяемые полюсами отношения типа агрегация. Поскольку на диаграмме классов множественности полюсов `part` и `ordr` заданы выражением `[1..*]`, их значения в графических символах объектов заданы как значения набора объектов в виде перечисления значений в фигурных скобках.

Рассмотрим еще один пример отображения диаграммы классов в диаграмму объектов. Ранее, при изучении отношения типа ассоциация, мы рассматривали пример диаграммы классов, которая воспроизведена на рис. 13.12.

Диаграмма классов на рис. 13.12 моделирует структуру системы, остающейся неизменной на протяжении всего времени ее существования. Однако,

в различные моменты времени в памяти компьютера может находиться различное количество связанных между собой объектов классов `Man` (мужчина) и `Child` (ребёнок).

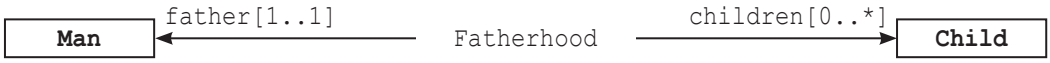



**Рис. 13.12.** Пример диаграммы классов с отношением типа ассоциация

На рис. 13.13 изображена диаграмма объектов, соответствующая диаграмме классов на рис. 13.12, которая отражает состояние этой системы для некоторого момента времени.

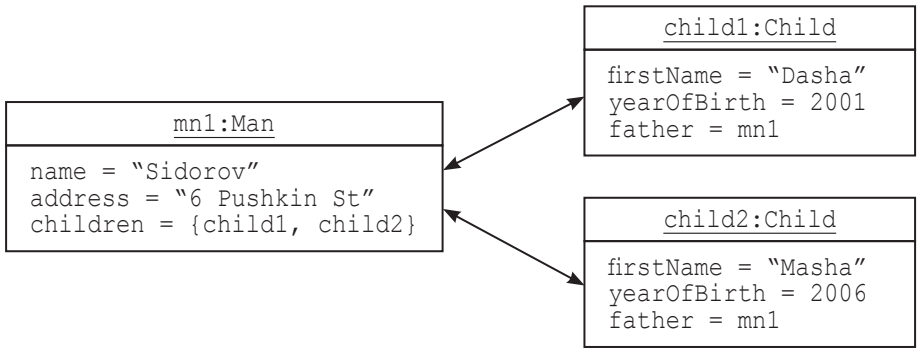



**Рис. 13.13.** Одна из возможных диаграмм объектов, соответствующих диаграмме классов на рис. 13.12

В диаграмму объектов, в отличие от диаграммы классов, включено некоторое количество полей и их значения, специфицирующие состояния объектов для заданного момента времени. Объект класса `Man` содержит значение поля `name` (фамилия) и значение поля `address` (адрес проживания), а объекты класса `Child` — значения полей `firstName` (имя) и `yearOfBirth` (год рождения). В списки полей объектов дополнительно включены поля `children` и `father`, определяемые полюсами отношения `Fatherhood`.

Ясно, что в другой момент времени диаграмма объектов, соответствующая диаграмме классов на рис. 13.12, может быть другой. На ней могут появиться новые объекты, если, например, у мужчины появился ещё один ребёнок. Изменения могут коснуться также состояния любого из объектов, поскольку в процессе функционирования системы изменяются значения их полей.

В завершение серии примеров диаграмм объектов рассмотрим, каким образом может выглядеть диаграмма объектов, если на соответствующей диаграмме классов отношение типа ассоциация представлено в виде класса. На рис. 13.14 приведена ещё одна диаграмма классов из ранее рассмотренных.

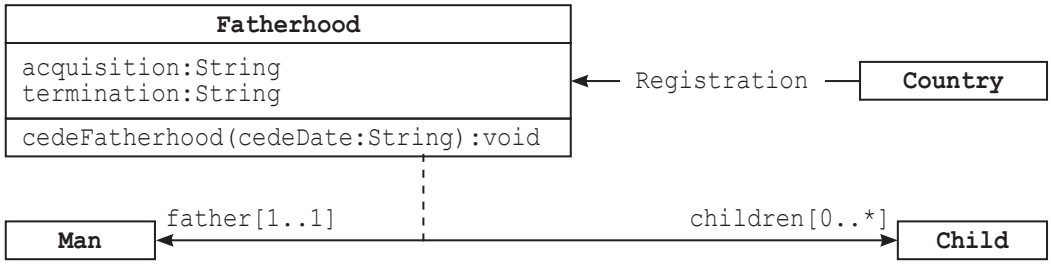



**Рис. 13.14.** Пример диаграммы классов с отношением типа ассоциация, представленным в виде класса

Модель, приведенная на диаграмме 13.14, является развитием модели на рис. 13.12 в направлении: (1) более полного описания отношения `Fatherhood` и (2) на расширение модели путём введения в ее структуру ещё одного класса с именем `Country` (страна). Прежде чем отображать диаграмму классов на рис. 13.14 в диаграмму объектов, представим ее так, как это показано на рис. 12.33. Новое представление диаграммы приведено на рис. 13.15. В новом представлении ассоциация-класс `Fatherhood` явно ассоциирован с классами `Man` и `Child`.

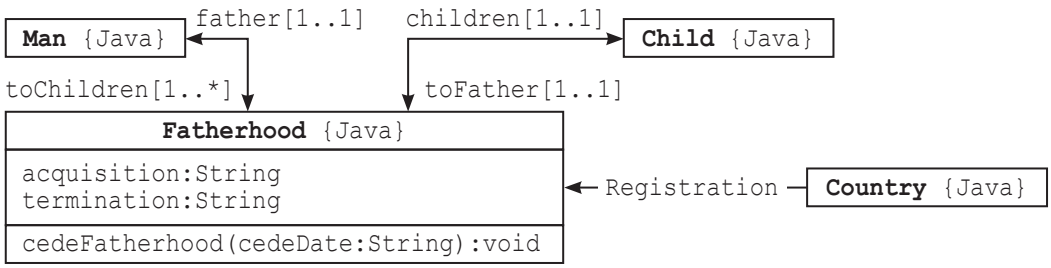



**Рис. 13.15.** Диаграмма классов, в которой ассоциация-класс `Fatherhood` явно ассоциирован с классами `Man` и `Child`

На рис. 13.16 приведен один из возможных вариантов диаграммы объектов, соответствующей диаграмме классов на рис. 13.15.

Диаграмма объектов, изображенная на рис. 13.16, отличается от диаграммы объектов, изображенной на рис. 13.13, тем, что в ней появились дополнительные объекты класса `Fatherhood`, при помощи которых в модель добавлены сведения об этом отношении. В частности, при помощи значения поля `acquisition` (дата регистрации) указана точная дата регистрации каждого ребенка. В модель также включены два анонимных объекта класса `Country`, при помощи которых можно специфицировать страну, в которой осуществлена регистрация отношения отцовства для каждого ребёнка. Как и в предыдущих примерах, в графические символы объектов включены поля, определяемые полюсами отношений типа агрегация с учетом направления навигации.

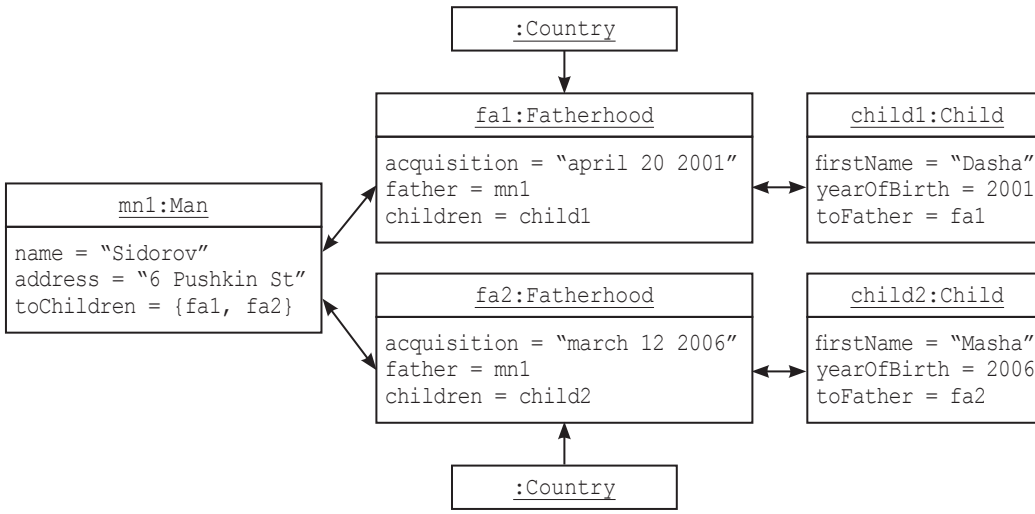



**Рис. 13.16.** Одна из возможных диаграмм объектов, соответствующих диаграмме классов на рис. 13.15

## Упражнения для практических занятий

13.1. Для моделирования структуры системы можно использовать диаграмму классов, диаграмму пакетов и диаграмму объектов. Сформулируйте правила, позволяющие разработчику модели сделать однозначный выбор одной из перечисленных диаграмм для моделирования структуры системы.

13.2. Проанализируйте типы отношений, которые используются при построении диаграммы классов, и рассмотрите возможность их применения для построения диаграммы пакетов. Объясните, какие типы отношений применимы для построения диаграммы пакетов и на каком основании вы пришли к этому выводу. Если вы считаете, что некоторые типы отношений не применимы в случае диаграммы пакетов, то объясните, почему вы так считаете.

13.3. Трансформируйте диаграмму классов, полученную в результате решения упражнения 4.3 (модель структуры университетской библиотеки) в диаграмму пакетов. Сформулируйте принципы группировки классов в пакеты.

13.4. Перечислите случаи, когда для доступа к классу, размещённому в пакете, необходимо указывать полное имя класса, и случаи, когда полное имя класса можно не указывать.

13.5. Приведите пример диаграммы объектов, которая соответствует диаграмме классов, полученной в результате решения упражнения 4.4 (модель структуры простого чертежа). Рассмотрите случай, когда чертеж состоит из одной окружности и одного прямоугольника, соединенных линией.

13.6. Приведите пример диаграммы объектов, которая соответствует диаграмме классов, полученной в результате решения упражнения 4.5 (модель ориентированного графа).

13.7. Приведите пример диаграммы объектов, которая соответствует диаграмме классов, полученной в результате решения упражнения 4.7 (модель двудольного ориентированного графа).

13.8. Приведите пример диаграммы объектов, которая соответствует диаграмме классов, полученной в результате решения упражнения 4.15 (модель системы, включающей книгу, её автора, издателя и распространителя). Рассмотрите случай, когда в памяти находится по одному объекту классов `Book`, `Author`, `Publisher` и два объекта класса `Distributor`.

# Литература для углубленного изучения


1. Else Lervik, Vegard B. Havdal. *Java the UML Way*. John Wiley. 2002.
2. Jos Warmer, Anneke Kleppe. *The Object Constraint Language. Second Edition*. Addison-Wesley. 2003.
3. Дж. Рамбо, М.Блаха. UML 2.0. *Объектно-ориентированное моделирование и разработка. 2-е издание.* СПб: Питер, 2007. — 574 с.
4. Крег Ларман. *Применение UML 2.0 и шаблонов проектирования. 3-е издание.* Вильямс. 2007.
5. Кей С. Хорстманн, Гари Корнелл. *Java 2. Библиотека профессионала. Core Java 2, Volume I — Fundamentals.* — 8-е изд. — М.: Вильямс, 2008. — Т. I: Основы. — 816 с.
6. Б. Эккель. *Философия Java.* — СПб: Питер, 2009. — 638 с.
7. Kishori Sharan. *Harnessing Java 7. A Comprehensive Approach to Learning Java, volume 1*. 2011.
8. Kishori Sharan. *Harnessing Java 7. A Comprehensive Approach to Learning Java, volume 2*. 2011.
9. Kishori Sharan. *Harnessing Java 7. A Comprehensive Approach to Learning Java, volume 3*. 2011.
10. Официальный сайт Oracle. *The Java Tutorials* [электронный ресурс]. Режим доступа: http://docs.oracle.com/javase/tutorial/index.html
11. Y.Daniel Liang. *Introduction to Java programming, Comprehensive Version*, 9th Edition, Pearson, 2012.
12. Paul J. Deitel, Harvey M. Deitel. *Java How to Program*, 9th Edition, 2012.