

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

КОВАЛЕНКО Л. Б., КУЗНІЧЕНКО С. Д.
АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ

Конспект лекцій

Одеса
Одеський державний екологічний університет
2015

УДК 681.3
К56

Рекомендовано методичною радою Одеського державного екологічного університету Міністерства освіти і науки України як конспект лекцій (протокол №2 від 24.09. 2015 р.)

Коваленко Л. Б., Кузніченко С. Д.

Алгоритмізація та програмування: конспект лекцій. Одеса, Одеський державний екологічний університет, 2015. 326с.

У навчальному посібнику викладено питання щодо моделювання і кодування методів і класів мовою програмування Java. Посібник орієнтований на читача, який не має попередніх знань і навичок в галузі програмування, і розрахований на студентів, які навчаються за спеціальністю «Комп'ютерні науки». Посібник може використовуватись у вищих і середніх закладах освіти при викладенні дисциплін, присвячених введенню у програмування і алгоритмізацію, а також основам об'єктно-орієнтованого програмування.

Конспект лекцій з навчальної дисципліни "Алгоритмізація та програмування" для студентів І курсу денної форми навчання, спеціальність – комп'ютерні науки.

ISBN 978-966-186-087-1

СОДЕРЖАНИЕ

1 ВВЕДЕНИЕ В ИНФОРМАТИКУ И ПРОГРАММИРОВАНИЕ	9
1.1 Архитектура компьютеров. Принципы фон Неймана.....	9
1.2 Арифметические основы компьютеров.....	14
1.2.1 Позиционные системы счисления	14
1.2.2 Перевод натуральных чисел из одной системы счисления в другую.....	17
1.2.3 Арифметические операции в разных системах счисления	19
1.3 Представление чисел в компьютере.....	21
1.3.1 Числа с фиксированной точкой	21
1.3.2 Выполнение арифметических операций над целыми числами	24
1.3.3 Числа с плавающей точкой.....	27
1.4 Команды компьютера	32
1.5 Классификация языков программирования	33
1.6 Этапы создания и подготовки программы к выполнению	35
1.7 Алгоритмы и основные алгоритмические структуры	41
1.7.1 Свойства и формы представления алгоритма.....	41
1.7.2 Основные алгоритмические структуры	44
Упражнения к главе 1	47
2 ЭЛЕМЕНТЫ ЯЗЫКА JAVA	50
2.1 Краткая история создания Java.....	50
2.2 Установка среды Java	51
2.3 Выполнение Java-программы	55
2.4 Некоторые интегрированные среды разработки	58
2.5 Лексические основы	60
2.5.1 Лексемы, комментарии и пробелы	60
2.5.2 Виды лексем.....	61
2.6 Примитивные типы данных и переменные	65
2.6.1 Переменные.....	66
2.6.2 Целые типы	67
2.6.3 Типы с плавающей точкой	69
2.6.4 Символьный тип	70
2.6.5 Логический тип.....	70
2.6.6 Приведение типов.....	70
2.7 Операции над данными	72
2.7.1 Арифметические операции.....	72
2.7.2 Автоматическое повышение типа в выражениях.....	74
2.7.3 Операции сравнения.....	75
2.7.4 Логические операции	76
2.7.5 Побитовые операции.....	77

2.7.6 Сдвиги.....	78
2.7.7 Условная операция.....	79
2.7.8 Оператор присваивания.....	80
2.7.9 Приоритет операций	82
2.7.10 Выражения	82
2.8 Методы класса Math	84
2.8.1 Тригонометрические функции.....	84
2.8.2 Экспоненциальные функции.....	85
2.8.3 Функции округления.....	86
2.9 Форматированный консольный ввод-вывод.....	87
2.9.1 Основы форматированного вывода данных разных типов.....	89
2.9.2 Специальные флаги формата	92
2.9.3 Консольный ввод с помощью класса Scanner	94
Упражнения к главе 2	97
3 УПРАВЛЯЮЩИЕ ОПЕРАТОРЫ.....	100
3.1 Операторы выбора	100
3.1.1 Условный оператор if.....	100
3.1.2 Вложенный оператор if.....	103
3.1.2 Оператор выбора switch.....	106
3.2 Операторы цикла.....	112
3.2.1 Цикл с предусловием while	113
3.2.2 Цикл с постусловием do-while	115
3.2.3 Цикл с параметром for.....	116
3.2.4 Разновидности цикла for.....	119
3.3 Операторы перехода	122
3.3.1 Оператор break.....	122
3.3.2 Оператор continue.....	124
3.3.3 Организация безусловного перехода в Java	125
3.4 Некоторые циклические алгоритмы и программы.....	127
3.4.1 Алгоритм определения простого числа	127
3.4.2 Алгоритм нахождения наибольшего общего делителя.....	129
3.4.3 Рекуррентные последовательности	131
Рис. 3.7 – Блок-схема алгоритма вычисления n-го числа Фибоначчи	134
3.4.4 Схема Горнера	136
3.4.5 Степенные ряды.....	137
Упражнения к главе 3.....	139
4 МАССИВЫ.....	142
4.1 Описание и характеристики одномерных массивов	142
4.2 Базовые алгоритмы обработки одномерных массивов.....	145
4.2.1 Ввод и вывод массива	146
4.2.2 Версия for-each цикла for.....	148

4.2.3	Вычисление сумм, количеств и произведений элементов массива	150
4.2.4	Поиск максимального и минимального элемента массива	157
4.2.5	Поиск элементов в массиве	162
4.2.6	Перестановка элементов массива	163
4.2.7	Вставка, удаление и циклический сдвиг элементов массива.....	166
4.2.8	Проверка соответствия элементов массива некоторому условию.	168
4.3	Сортировка массива.....	171
4.3.1	Сортировка методом пузырька	172
4.3.2	Сортировка методом выбора.....	174
4.3.3	Сортировка методом вставки	178
4.4	Работа с последовательностями	181
4.5	Многомерные массивы.....	183
4.5.1	Описание многомерных массивов	184
4.5.2	Базовые алгоритмы обработки двумерных массивов.....	186
4.5.3	Сортировка двумерных массивов	193
	Упражнения к главе 4	195
5	ВВЕДЕНИЕ В КЛАССЫ, ОБЪЕКТЫ И МЕТОДЫ	198
5.1	Описание класса.....	198
5.2	Области видимости переменных.....	200
5.3	Создание объектов.....	201
5.4	Методы класса.....	204
5.4.1	Объявление метода.....	205
5.4.2	Вызов метода	208
5.5	Статические элементы.....	209
5.6	Способы передачи аргументов	210
5.7	Список аргументов переменной длины.....	214
5.8	Конструкторы.....	216
5.9	Класс Stack.....	219
	Упражнения к главе 5	221
6	РЕКУРСИЯ.....	224
6.1	Стек вызовов.....	224
6.2	Рекурсия без ветвления	224
6.3	Рекурсия с ветвлением	228
6.4	Косвенная рекурсия	229
6.5	Рекурсивный поиск в массивах	231
6.6	Быстрая сортировка	234
6.7	Сортировка слиянием	236
6.8	Стандартные средства Java для работы с массивами.....	239
	Упражнения к главе 6	242
7	СТРОКИ И ТЕКСТОВЫЙ ВВОД-ВЫВОД	244
7.1	Класс String.....	244

7.1.1 Конструкторы класса String	244
7.1.2 Интернирование строк	246
7.1.3 Нахождение длины строки и выбор символов из строки	247
7.1.4 Сравнение строк	249
7.1.5 Поиск символов и подстроки в строке.....	255
7.1.6 Извлечение подстроки из строки.....	257
7.1.7 Модификация, замена и разбиение строк	259
7.1.8 Преобразование между строками и массивами	261
7.1.9 Преобразование символов и чисел в строки	261
7.2 Классы StringBuilder и StringBuffer	263
7.2.1 Конструкторы класса StringBuilder	263
7.2.2 Добавление, удаление и замена подстроки	266
7.3 Класс Character	269
7.4 Аргументы командной строки.....	272
7.5 Синтаксический разбор строки	273
7.5.1 Метод split() класса String	273
7.5.2 Класс StringTokenizer	276
7.6 Текстовый ввод-вывод	277
7.6.1 Класс File	279
7.6.2 Файловый ввод - вывод	284
Упражнения к главе 7	292
8 ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ	298
8.1 Пример реализации динамического массива	298
8.2 Абстрактные типы данных	300
8.3 Стеки	301
8.3.1 Стек фиксированной емкости	302
8.3.2 Стек с переменным размером массива.....	304
8.4 Двухстековый алгоритм Дейкстры для вычисления выражения.....	309
8.5 Очередь	313
8.6 Связные списки	316
Упражнения к главе 8	327

1 ВВЕДЕНИЕ В ИНФОРМАТИКУ И ПРОГРАММИРОВАНИЕ

1.1 Архитектура компьютеров. Принципы фон Неймана

В данном разделе рассмотрим классическую архитектуру компьютеров и общие принципы их построения.

Программное обеспечение не может быть выполнено без применения компьютера, поэтому прежде чем говорить о программировании, необходимо получить представление об основных компонентах, составляющих компьютерную систему, их характеристиках, свойствах и принципах работы. Ограничимся лишь теми свойствами, которые необходимы программистам и важны для понимания рассматриваемых в дальнейшем тем. Более глубокие знания по устройству компьютера можно получить из специальных курсов, например, «Компьютерная схемотехника и архитектура компьютеров».

Компьютер (англ. computer – «вычислитель») представляет собой программируемое электронное устройство, способное выполнять конечное множество арифметических и логических операций в соответствии с заданной последовательностью команд. Эти наборы команд называются *компьютерными программами*. Совокупность команд, выполняемых конкретным компьютером, называется *системой команд* этого компьютера.

В основу построения большинства компьютеров положены следующие общие принципы, сформулированные в 1946 году американским ученым Джоном фон Нейманом.

Принцип двоичного кодирования

Для записи команд и данных в компьютере используется *двоичная система счисления*. В двоичной системе счисления есть только две цифры: 0 и 1. Удобство использования двоичной системы в вычислительной технике обусловлено тем, что ее можно легко реализовать, так как для этого нужны технические устройства всего с двумя устойчивыми состояниями (есть ток – нет тока, намагничен – не намагничен и т.п.). Представление информации посредством только двух состояний надежно и помехоустойчиво. Двоичная арифметика намного проще десятичной, для выполнения логических преобразований возможно применение аппарата булевой алгебры. Одна двоичная цифра называется *битом* (англ. binary digit – двоичная цифра). Последовательность из восьми бит, принято называть *байтом*. Один байт – это единица объема памяти. Память компьютера измеряется в байтах, но чаще для этого используются такие единицы, как килобайт ($1\text{Кбайт} = 2^{10}$ байт = 1024 байт), мегабайт ($1\text{Мбайт} = 2^{20}$ байт = 1 048 576 байт) и гигабайт ($1\text{Гбайт} = 2^{30}$ байт = 1 073 741 824 байт).

Принцип последовательного программного управления

Из него следует, что программа состоит из набора команд, которые выполняются процессором автоматически друг за другом в определенной последовательности. Если же нужно после выполнения команды перейти не к следующей, а к какой-то другой, используются команды условного или

безусловного переходов. Они также позволяют выполнять определенную последовательность команд многократно.

Принцип однородности памяти

Важный принцип, в соответствии с которым память компьютера используется не только для хранения данных, но и программ. Команды и данные хранятся в одной и той же памяти. При этом и команды программы и данные кодируются в двоичной системе счисления, т.е. их способ записи одинаков. Поэтому в определенных ситуациях над командами можно выполнять те же действия, что и над данными. А это значит, что программа в процессе своего выполнения также может подвергаться преобразованию – команды одной программы могут быть получены как результаты исполнения другой программы. На этом принципе основаны методы трансляции – перевода текста программы с языка программирования высокого уровня в последовательность команд конкретного компьютера.

Принцип адресуемости памяти

Структурно основная память состоит из ячеек, каждая из которых имеет свой номер – *адрес*. Размер ячейки памяти обычно равен одному байту. Процессору в произвольный момент времени доступна любая ячейка. Выполняя обработку информации, он может по адресу найти в памяти нужную ячейку, прочитать из нее содержимое, выполнить необходимые действия и записать результат в другую ячейку памяти. Этот принцип дал возможность использовать переменные в программировании.

На основе перечисленных выше принципов строились первые поколения компьютеров. В более поздних поколениях происходили некоторые изменения, хотя принципы фон Неймана актуальны и сегодня.

Все компьютеры, построенные в соответствии с описанными принципами, относятся к компьютерам с *фоннеймановской архитектурой*. На рис. 1.1 приведена схема машины Фон Неймана¹. Толстыми (двойными) стрелками показаны потоки команд и данных, а тонкими – передача между отдельными устройствами компьютера управляющих и информационных сигналов.

Как видно из приведённого рисунка, машина Фон Неймана состоит из памяти, арифметико-логического устройства, устройства управления и устройств ввода/вывода.

Арифметико-логическое устройство (АЛУ) выполняет основную работу по переработке информации, хранимой в оперативной памяти. Производит арифметические и логические операции, а также операции сдвига. Обеспечивает обработку входных данных и формирование результата. К функциям АЛУ также относится формирование признака результата выполнения операции над данными, указывающей на равенство результата операции нулю, правильность выполнения операции и др. На

¹ Здесь рассматривается несколько модифицированная схема машины фон Неймана. На рисунке устройства ввода/вывода обмениваются данными непосредственно с памятью, а в схеме самого фон Неймана этот обмен производился через арифметико-логическое устройство (АЛУ) под контролем устройства управления.

основании данного признака компьютер может производить операции ветвления программы в зависимости от конкретных значений данных.

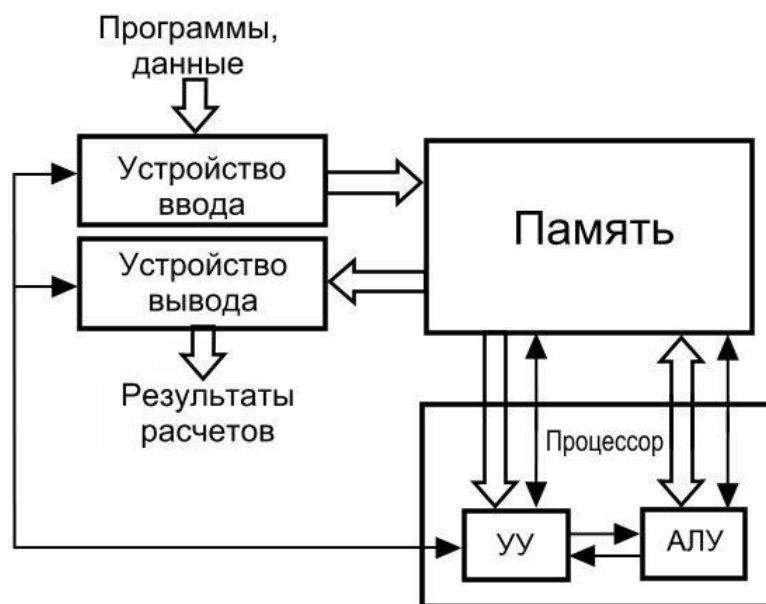


Рис.1.1 – Структура машины фон Неймана

Устройство управления (УУ) управляет всеми остальными устройствами компьютера. Обратите внимание, что УУ является единственным устройством, от которого на рис. 1.1 отходят тонкие стрелки управляющих сигналов ко всем другим устройствам. УУ вырабатывает распределенную во времени и пространстве последовательность внутренних и внешних управляющих сигналов, обеспечивающих выборку и выполнение команд. Устройство управления выполняет команды, по определённому правилу выбирая их из оперативной памяти, по очереди одна за другой. На этапе выполнения команды, в соответствии с типом реализуемой операции, УУ формирует требуемый набор команд низкого уровня для АЛУ и других устройств. Эти команды задают последовательность простейших низкоуровневых операций, таких, как пересылка данных, сдвиг данных, установка и анализ признаков, запоминание результатов и др. Кроме этого, УУ формирует синхронизирующие сигналы, координирующие действия внешних и внутренних устройств компьютера.

Устройство управления и АЛУ являются составной частью *центрального процессора* (Central Processing Unit, CPU) – основного рабочего компонента компьютера, управляющего вычислительным процессом и координирующего работу всех его устройств. Характеристики процессора непосредственно влияют на производительность компьютера.

Память хранит информацию, передаваемую из других устройств, в том числе поступающую в компьютер извне через устройство ввода, и выдает во все другие устройства информацию, необходимую для протекания

вычислительного процесса. Память разделяют на *внутреннюю* и *внешнюю*. В состав внутренней памяти входят основная память, кэш-память и регистры процессора. *Основная память* делится на постоянную и оперативную память.

Регистры процессора – это внутренняя память процессора, наиболее быстродействующая, но наименьшая по объему разновидность памяти. Обычно у процессора не более нескольких десятков регистров и каждый имеет свое определенное назначение.

Постоянная память (Read Only Memory, ROM) – постоянное запоминающее устройство. Данные, хранящиеся в ROM, нельзя изменять, они доступны только для чтения. Постоянная память предназначена для хранения программ, которые должны быть доступны компьютеру сразу после включения, еще до загрузки операционной системы. Например, в постоянной памяти хранится программа первоначального тестирования – базовая система ввода-вывода BIOS (англ. Basic Input Output System) компьютера.

Оперативная память (Random Access Memory, RAM) – это быстрое запоминающее устройство не очень большого объема, непосредственно связанное с процессором и предназначенное для записи, считывания и хранения выполняемых программ и данных. Оперативная память используется только для временного хранения данных и программ и не может сохранять свое содержимое после выключения питания компьютера.

Доступ к элементам оперативной памяти прямой – это означает, что каждая ячейка памяти имеет свой индивидуальный *адрес*. В любую ячейку памяти другие устройства (по толстым стрелкам на рис. 1.1) могут записать и считать информацию. Ячейки памяти нумеруются от нуля. Каждая ячейка состоит из более мелких частей, именуемых разрядами и нумеруемых также от нуля и до определенного числа. Количество разрядов в ячейке обозначает разрядность памяти. Каждый разряд хранит одну двоичную цифру или один бит информации. Число, содержащееся в ячейке, называют ее *значением* или *содержимым*. Из каждой ячейки памяти можно считать копию ее значения и передать в другое устройство компьютера, при этом первоначальное содержимое ячейки не меняется. При записи в память старое содержимое ячейки пропадает и заменяется новым. В этом случае говорят, что ячейке было *присвоено* новое значение.

Кэш-память (англ. cache) – очень быстрое запоминающее устройство небольшого объема, которое используется при обмене данными между процессором и оперативной памятью для согласования их скоростей работы. Скорость обработки информации процессором выше, чем менее быстродействующей оперативной памяти. Кэш-памятью управляет специальное устройство – контроллер, который, анализируя выполняемую программу, пытается предвидеть, какие данные и команды вероятнее всего понадобятся в ближайшее время процессору, и подкачивает их в кэш-память. Если же требуемая информация в кэш отсутствует, то процессор считывает ее непосредственно из оперативной памяти.

Внешняя память относится к внешним устройствам компьютера и предназначена для долгосрочного хранения данных и программ больших объемов. Внешняя память содержит разнообразные виды запоминающих устройств: накопители на жестких и гибких магнитных дисках, накопители на оптических дисках CD-ROM и CD-RW (англ. Compact Disk Read Only Memory – компакт-диск только для чтения, Compact Disk Read Write Memory – компакт-диск для чтения и записи), DVD-ROM (англ. Digital Video Disk – цифровой видеодиск), флеш-память.

Внешние устройства ввода-вывода обеспечивают взаимодействие компьютера с пользователями, объектами управления и другими компьютерами. Существует много разнообразных устройств, которые могут взаимодействовать с процессором и памятью. Часть этих устройств может быть встроена внутрь корпуса компьютера, часть вынесена за его пределы и общаться с компьютером через различные линии связи. К устройствам ввода информации можно отнести клавиатуру, мышь, сканер, графический планшет, сенсорный экран и т.д. Принтер, дисплей, графопостроитель – примеры *устройств вывода* информации.

Архитектура современных компьютеров предусматривает наличие специальных каналов прямого доступа к оперативной памяти для обмена данными с устройствами ввода-вывода, которые именуются *портами*. С точки зрения пользователя, порт – это интерфейс (разъем) для подключения устройства ввода-вывода к компьютеру.

Что именно должны делать устройства, приняв информацию через свой порт, и каким именно образом они должны поставлять информацию для чтения из порта, определяется электронными схемами устройств, получившими название *контроллеров*. Контроллеры – специализированные процессоры, выполняя большинство функций по управлению устройствами ввода-вывода, они тем самым разгружают центральный процессор и способствуют повышению его производительности.

Основной интерфейсной системой компьютера, обеспечивающей сопряжение и связь всех его устройств между собой, являются *шины*. Шины обеспечивают три направления передачи информации:

- между процессором и оперативной памятью;
- между процессором и портами ввода-вывода внешних устройств;
- между оперативной памятью и портами ввода-вывода внешних устройств (в режиме прямого доступа к памяти).

Физически шины состоят из большого числа параллельных металлических проводников, т.е. представляют собой многопроводные линии с гнездами для подключения электронных схем. Вся совокупность проводов шины разделяется на отдельные группы: *шину адреса*, *шину данных* и *шину управления*.

По шине данных передаются данные между различными устройствами. Например, считанные из оперативной памяти данные могут быть переданы процессору для обработки, а затем полученные данные могут быть отправлены обратно в оперативную память для хранения. Таким образом,

данные по шине данных могут передаваться от устройства к устройству в любом направлении. Количество линий, выделенных для передачи данных, называют шириной шины данных. Ширина определяется разрядностью процессора, т.е. количеством битов, которые процессор обрабатывает за один такт². Разрядность процессоров постоянно увеличивалась по мере развития компьютерной техники.

Выбор устройства или ячейки памяти, куда пересылаются или откуда считываются данные по шине данных, производит процессор. Каждое устройство (регистры процессора, порты ввода-вывода) или ячейки оперативной памяти имеют свой адрес. Адрес передается по шине адреса, причем сигналы по ней передаются в одном направлении от процессора к оперативной памяти и устройствам (однонаправленная шина). Ширина шины адреса определяет адресное пространство процессора, т.е. количество ячеек оперативной памяти, которые могут иметь уникальные адреса. Использование отдельных шин для адреса и данных повышает производительность компьютера, особенно во время записи в память, так как данные и адреса ячеек передаются параллельно.

По шине управления передаются сигналы, определяющие характер обмена информацией. Сигналы управления определяют, какую операцию считывание или запись информации из памяти нужно производить, синхронизируют обмен информацией между устройствами, передают сигналы прерывания и т.п.

Вся информация, с которой работает компьютер, кодируется числами. Независимо от того, графическая, текстовая или звуковая эта информация, что бы ее мог обрабатывать процессор она должна тем или иным образом быть представлена числами. Познакомимся подробнее с тем, как представляются числа в различных системах счисления и в компьютере.

1.2 Арифметические основы компьютеров

1.2.1 Позиционные системы счисления

Система счисления (англ. Number System) – это совокупность приемов и правил, по которым числа записываются и читаются. Существуют *позиционные* (англ. Positional Number System) и *непозиционные* (англ. Non-positional Number System) системы счисления.

Непозиционная система счисления – это система счисления, в которой вес цифры, т.е. тот вклад, который она вносит в значение числа, не зависит от ее позиции. Примером непозиционной системы счисления является римская система (римские цифры).

В позиционных системах счисления вес каждой цифры изменяется в зависимости от ее положения, т.е. позиции в последовательности цифр.

² Такт – промежуток времени, в течение которого процессором может быть выполнена одна элементарная операция.

Рассмотрим, число 535,51 цифра 5 здесь обозначает пять сотен, пять единиц и пять десятых долей единицы. Разложим данное число на разрядные слагаемые:

$$535,51 = 500 + 30 + 5 + 0,5 = 5 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0 + 5 \cdot 10^{-1} + 1 \cdot 10^{-2}$$

Любая позиционная система характеризуется своим основанием. *Основание позиционной системы счисления* – это количество различных цифр, используемых для отображения чисел в данной системе счисления. Рассмотренный выше пример иллюстрирует запись числа в привычной для нас *десятичной системе счисления*.

Возможно бесконечное множество позиционных систем счисления: двоичная, троичная, четверичная и т.д.

Примем некоторое натуральное число q за основание системы счисления, тогда значение числа N в q -ой системе счисления будет равно значению следующего полинома:

$$N = \sum_{i=-m}^n a_i q^i = a_n q^n + a_{n-1} q^{n-1} + \dots + a_1 q^1 + a_0 q^0 + a_{-1} q^{-1} + \dots + a_{-m} q^{-m}, \quad (1.1)$$

где a_i – значение цифры в i -ом разряде числа;

q^i – вес i -ого разряда числа;

$i = -1, -2, \dots, -m$ – разряды дробной части числа;

$i = 0, 1, \dots, n-1$ – разряды целой части числа.

Рассмотрим более подробно позиционные системы счисления, которые используются в программировании. Ранее уже отмечалось, что двоичная система счисления сделала возможным компьютерную революцию, поскольку открыла простой и общий способ представления информации. К недостаткам двоичной системы следует отнести быстрый рост двоичных разрядов, который необходим для записи чисел. Двоичная система счисления удобна для компьютера, но не совсем удобна для человека, т.к. она громоздкая и ее запись не привычна. Для сокращения записи двоичных чисел в программировании также применяется *восьмеричная* и *шестнадцатеричная системы счисления*. Для записи чисел в восьмеричной системе используют 8 цифр: 0,1,2,3,4,5,6,7. Для записи чисел в шестнадцатеричной системе используют 16 цифр: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. Представление целых чисел в различных системах счисления приведено в табл.1.1.

Так как $8 = 2^3$, а $16 = 2^4$, то для записи одного и того же числа требуется в 3 раза меньше разрядов (в восьмеричной) и в 4 раза меньше разрядов (в шестнадцатеричной) системах, чем в двоичной системе счисления.

В каждой позиционной системе счисления цифры упорядочены в соответствии со своим значением: $1 > 0$, $2 > 1$, и т.д.

Продвижением (сдвигом) цифры называют замену ее следующей по величине цифрой. Продвинуть цифру 1 значит заменить её на 2, продвинуть цифру 2 значит заменить её на 3 и т.д. Продвижение старшей цифры (например, цифры 9 в десятичной системе) означает замену её на 0. В двоичной системе, использующей только две цифры – 0 и 1, продвижение 0 означает замену его на 1, а продвижение 1 – замену её на 0.

Таблица 1.1 – Соответствие цифр некоторых систем счисления*

Основание системы счисления			
2	8	10	16
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000		8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F
10000	20	16	10

**(серые ячейки – цифры системы счисления, белые – числа)*

Операция продвижения цифр позволяет получить следующее по величине число. Так, для образования целого числа, следующего за любым данным целым числом, нужно продвинуть самую правую цифру числа; если какая-либо цифра после продвижения стала нулем, то нужно продвинуть цифру, стоящую слева от неё.

В двоичной системе счисления, после 1 идет число 10. Выполним продвижение 1. Самая правая (младшая) цифра числа станет равной 0, а единица уйдет вперед (1→10). Выполняя данное правило, запишем первые десять чисел в двоичной системе: 0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001.

1.2.2 Перевод натуральных чисел из одной системы счисления в другую

Перевод из восьмеричной и шестнадцатеричной системы в двоичную систему. Рассмотрим простой способ перевода восьмеричных и шестнадцатеричных чисел в двоичную систему счисления. Поскольку $16=2^4=10000_2$, а $8=2^3=1000_2$, то одна шестнадцатеричная цифра используется для изображения четырех двоичных битов, а одна восьмеричная цифра – для изображения трех двоичных битов³. С учетом этого, для перевода восьмеричных и шестнадцатеричных чисел в двоичную систему достаточно каждую цифру заменить эквивалентной ей двоичной триадой (тройкой цифр) или тетрадой (четверкой цифр).

Ниже приведены примеры переводов из восьмеричной и шестнадцатеричной систем счисления в двоичную систему.

$$\begin{aligned} 327,5_8 &= \underbrace{011}_3 \underbrace{010}_2 \underbrace{111}_7, \underbrace{101}_5_2 \\ 9A8,C_8 &= \underbrace{1001}_9 \underbrace{1010}_A \underbrace{1000}_8, \underbrace{1102}_C_2 \end{aligned}$$

Перевод из двоичной в восьмеричную и шестнадцатеричную системы счисления. Чтобы перевести число из двоичной системы в восьмеричную или шестнадцатеричную, его нужно разбить влево и вправо от запятой на триады (для восьмеричной) или тетрады (для шестнадцатеричной) и каждую такую группу заменить соответствующей восьмеричной (шестнадцатеричной) цифрой.

$$\begin{aligned} & \begin{array}{c} 11|010|001,|011|11_2 = 321,36_8 \\ \hline \underbrace{011}_3 \underbrace{010}_2 \underbrace{001}_1 \underbrace{011}_3 \underbrace{110}_6 \end{array} \\ & \begin{array}{c} 1101|0001,|0111|1_2 = D1,78_{16} \\ \hline \underbrace{1101}_{13} \underbrace{0001}_1 \underbrace{0111}_7 \underbrace{1000}_8 \end{array} \\ & (D_{16}) \end{aligned}$$

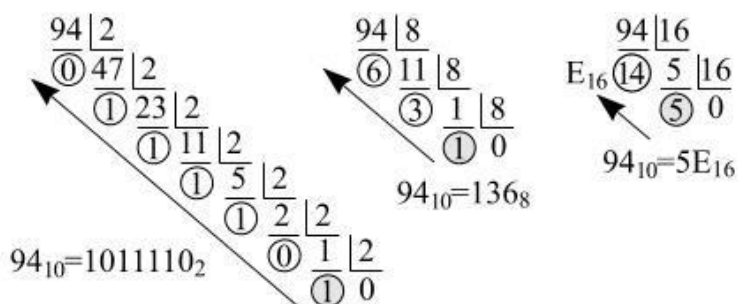
Перевод чисел из десятичной системы в любую другую

Для чисел, имеющих как целую, так и дробную части, перевод из десятичной системы счисления в другую осуществляется отдельно для целой и дробной частей по правилам, указанным ниже.

Для перевода целого числа производят последовательное целочисленное деление на основание системы, в которую переводят, сначала самого числа, затем полученные частные. Процесс заканчивается, когда

³ В дальнейшем, чтобы отличить в какой системе счисления записано число, будем указывать основание в виде индекса в десятичной системе счисления, например, 1000_2 .

Проиллюстрируем это правило примером, в котором переведем в двоичную, восьмеричную и шестнадцатеричную системы счисления десятичное число 94.



Умножение производится до тех пор, пока дробная часть произведения не станет равной нулю либо до заданной точности. Если требуемая точность перевода числа составляет k знаков после запятой, то предельная абсолютная погрешность при этом будет равна выражению: $q^{-(k+1)}/2$.

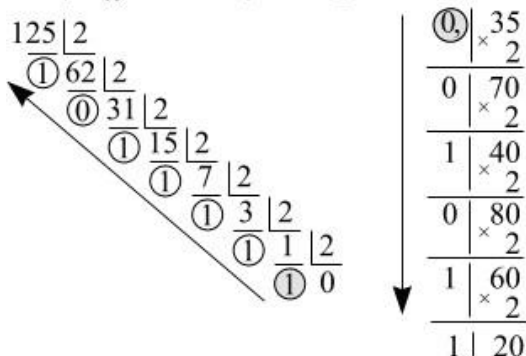
Проиллюстрируем это правило примером, в котором переведем в двоичную, восьмеричную и шестнадцатеричную системы счисления десятичное число 0,94.



Рассмотрим пример перевода из десятичной системы счисления в другую систему числа, имеющего как целую, так и дробную части.

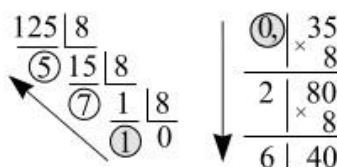
Двоичная система счисления

$$125,35_{10} = 1111101,01011_2$$



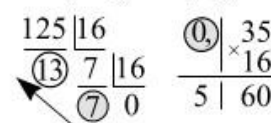
Восьмеричная система счисления

$$125,35_{10} = 175,26_8$$



Шестнадцатеричная система счисления

$$125,35_{10} = 7D,5_{16}$$



Перевод чисел в десятичную систему счисления

Перевод чисел в десятичную систему счисления производится с использованием записи чисел по основанию системы, и проведения арифметических операций в десятичной арифметике. Перевод сводится к вычислению значения многочлена (1.1).

Разряды 4 3 2 1 0 -1 -2

$$\text{Число } 10011,11_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 19,75_{10}$$

Разряды 2 1 0 -1

$$\text{Число } 732,4_8 = 7 \cdot 8^2 + 3 \cdot 8^1 + 2 \cdot 8^0 + 4 \cdot 8^{-1} = 474,75_8$$

Разряды 2 1 0

$$\text{Число } 5DF_{16} = 5 \cdot 16^2 + 13 \cdot 16^1 + 15 \cdot 16^0 = 1503_{16}$$

1.2.3 Арифметические операции в разных системах счисления

Основными арифметическими операциями, применяемыми в системе команд любого компьютера, являются *сложение*, *вычитание*, *умножение* и *деление*. Правила выполнения этих операций в десятичной системе хорошо известны – это сложение, вычитание, умножение столбиком и деление углом. Эти правила применимы и ко всем другим позиционным системам счисления.

Сложение. При сложении, цифры суммируются по разрядам. Если при этом возникает избыток, то он переносится влево. В качестве результата в соответствующем разряде записывается остаток от деления, полученной суммы разрядов на основание системы счисления, а в старший разряд переносится частное.

Вычислим сумму двух десятичных чисел 15_{10} и 6_{10} , а также их двоичных, восьмеричных и шестнадцатеричных эквивалентов – 1111_2 и 110_2 , 17_8 и 6_8 , F_{16} и 6_{16} .

$$\begin{array}{rcl}
 15_{10} = 1111_2 & 15_{10} = 17_8 & 15_{10} = F_{16} \\
 6_{10} = 110_2 & 6_{10} = 6_8 & 6_{10} = 6_{16}
 \end{array}$$

$$\begin{array}{r}
 + 15_{10} \\
 + 6_{10} \\
 \hline
 21_{10}
 \end{array}
 \quad
 \begin{array}{r}
 + 1111_2 \\
 + 0110_2 \\
 \hline
 10101_2
 \end{array}
 \quad
 \begin{array}{r}
 + 17_8 \\
 + 6_8 \\
 \hline
 25_8
 \end{array}
 \quad
 \begin{array}{r}
 + F_{16} \\
 + 6_{16} \\
 \hline
 15_{16}
 \end{array}$$

$7+6=13=8+5$ $15+6=21=16+5$

Вычитание. Вычитание в любой системе счисления выполняется по тем же правилам, что и в десятичной. Ниже приведен пример, иллюстрирующий вычитание 1 от чисел 100_2 , 100_8 и 100_{16} .

$$\begin{array}{r}
 - 100_2 \\
 - 1_2 \\
 \hline
 11_2
 \end{array}
 \quad
 \begin{array}{r}
 - 100_8 \\
 - 1_8 \\
 \hline
 77_8
 \end{array}
 \quad
 \begin{array}{r}
 - 100_{16} \\
 - 1_{16} \\
 \hline
 FF_{16}
 \end{array}$$

Умножение. Выполняя умножение многозначных чисел в различных позиционных системах счисления, можно использовать обычный алгоритм перемножения чисел в столбик, но при этом результаты перемножения и сложения однозначных чисел необходимо заимствовать из соответствующих рассматриваемой системе таблиц умножения и сложения. Ввиду чрезвычайной простоты таблицы умножения в двоичной системе, умножение в данной системе счисления сводится лишь к сдвигам множимого и сложениям. Рассмотрим пример умножения двоичных чисел при $n=3$:

$$\begin{array}{r}
 \times 101_2 \\
 110_2 \\
 + 101_2 \\
 101_2 \\
 \hline
 11110_2
 \end{array}$$

$$101_2 = 5_{10}, \quad 110_2 = 6_{10}, \quad 5 \times 6 = 30_{10},$$

$$11110_2 = 2^4 + 2^3 + 2^2 + 2^1 = 16 + 8 + 4 + 2 = 30_{10}.$$

Деление. Деление в любой позиционной системе счисления происходит по тому же алгоритму, что и деление под углом в десятичной системе счисления. Наиболее просто это делать в двоичной системе счисления.

$$\begin{array}{r}
 - 11110_2 \overline{) 110_2} \\
 \underline{110_2} \\
 - 00110_2 \\
 \underline{110_2} \\
 0_2
 \end{array}$$

$$11110_2 = 30_{10}, 110_2 = 6_{10},$$

$$30 \div 6 = 5,$$

$$101_2 = 5_{10}.$$

1.3 Представление чисел в компьютере

Разработка прикладного программного обеспечения требует от программиста знаний о том, каким образом представляются числовые данные в компьютере. Для решения большинства задач чаще всего используются целые и вещественные числа. При записи в компьютере целочисленных данных число переводится в двоичную систему и записывается в прямом коде. Диапазон представляемых чисел в этом случае ограничивается количеством выделенных для записи разрядов. Для вещественных данных обычно используются две формы записи: число с фиксированной точкой (англ. fixed point) и число с плавающей точкой (англ. floating point).

Отображение внешней информации во внутреннее представление называется кодированием. *Кодом* (франц. code, от лат. codex — свод законов) называют как сам способ отображения, так и множество слов (кодовых комбинаций), используемых при кодировании.

1.3.1 Числа с фиксированной точкой

Число с фиксированной точкой – это формат представления числа с неизменным расположением точки, которая отделяет целую часть числа от дробной (рис.1.2). Другими словами, фиксированная точка указывает на то, сколько и какие разряды машинного слова отведены под изображение целой и дробной частей числа. Запятая в разрядной сетке может быть зафиксирована, в принципе, после любого разряда.

Запись числа с фиксированной точкой обычно имеет знаковый и цифровые разряды. Знаковый разряд содержит информацию о знаке числа. Он равен нулю для положительных чисел и единице – для отрицательных. Цифровые разряды – это разряды, в которых расположены значащие цифры числа.

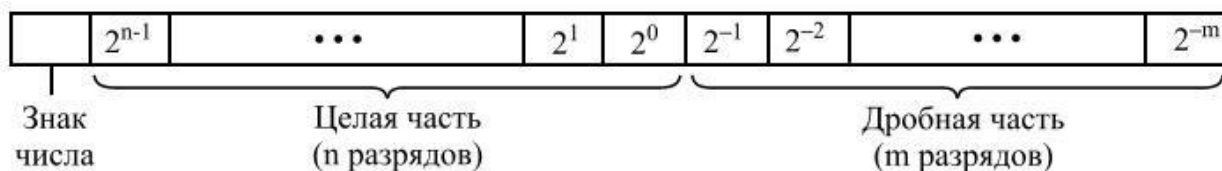


Рис.1.2 – Ячейки с целой и дробной частью

Запись целого числа можно рассматривать как частный случай записи числа с фиксированной точкой, при которой все разряды, кроме знакового, используются под целую часть (рис.1.3).

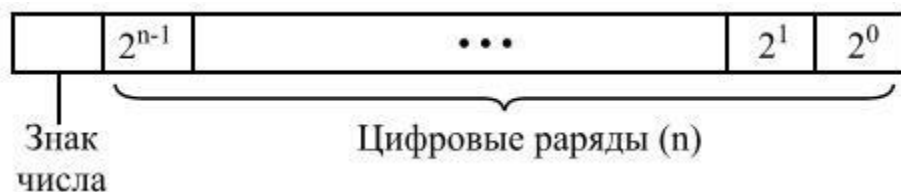


Рис.1.3 – Ячейка с записью целого числа

При работе на компьютере, числа с фиксированной запятой могут вводиться и отображаться в любом виде, однако при сохранении в память для упрощения операций над ними, числа преобразуются в соответствии с разрядной сеткой и хранятся либо с запятой, фиксированной после последнего разряда (целые числа), либо с запятой перед старшим цифровым разрядом (правильные дроби, по модулю меньше единицы). В современных компьютерах режим с фиксированной точкой используется преимущественно для представления целых чисел.

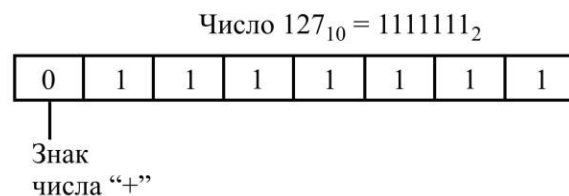
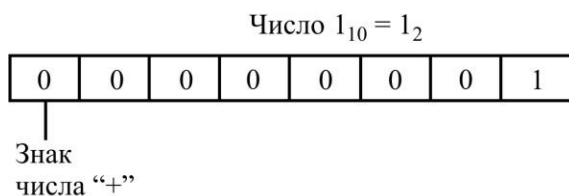
Обычно целые числа занимают один, два, четыре или восемь байт памяти. Диапазоны значений целых чисел (без знака и со знаком) в разных форматах приведены в табл.1.2.

К достоинствам использования чисел с фиксированной точкой относятся простота выполнения арифметических операций и высокая точность изображения чисел. К недостаткам – небольшой диапазон представления величин.

Таблица 1.2 – Диапазоны значений целых чисел в различных форматах представления

Целые числа	Формат представления	Диапазон величин
без знака	1 байт	$0 \dots 2^8 - 1$ (0...255)
	2 байта	$0 \dots 2^{16} - 1$ (0...65 535)
со знаком	1 байт	$-2^7 \dots 2^7 - 1$ (-128...127)
	2 байта	$-2^{15} \dots 2^{15} - 1$ (-32 768...32 767)
	4 байта	$-2^{31} \dots 2^{31} - 1$ (-2 147 483 648...-2 147 483 647)

Для представления знаковых целых чисел в компьютере используются прямой, обратный и дополнительный коды. Положительные числа в прямом, обратном и дополнительном кодах изображаются одинаково – цифровая часть содержит двоичную запись числа, в знаковом разряде содержится 0. Например, для $n = 8$:



Диапазон представимых чисел: $0..2^{n-1}-1$

Отрицательные числа в прямом, обратном и дополнительном кодах имеют разное изображение.

Прямой код (англ. True code) отрицательного числа можно получить, записав в знаковый разряд цифру 1, а в разряды цифровой части числа – двоичный код его абсолютной величины. Например, для $n = 8$:



Диапазон представимых чисел: $-(2^{n-1}-1)..0$

Обратный код (англ. Inverted code) получается инвертированием всех цифр двоичного кода абсолютной величины числа, включая разряд знака: нули заменяются единицами, а единицы – нулями. Например:



Диапазон представимых чисел: $-(2^{n-1}-1)..0$

Дополнительный код (англ. Compliment code) получается образованием обратного кода с последующим прибавлением единицы к самому его младшему разряду. Например, для $n = 8$:



Диапазон представимых чисел: $-2^{n-1}..-1$

Заметим, что ноль имеет два представления в прямом и обратном коде, а в дополнительном коде представление нуля единственно.

Рассмотрим, как можно осуществить вычисление обратного и дополнительного кодов. Пусть x – число со знаком. Тогда числовое значение его обратного и дополнительного кодов можно определить с помощью функций $inv(x)$ и $comp(x)$:

$$inv(x) = \begin{cases} x, & \text{если } x \geq 0 \\ 2^n - 1 - |x|, & \text{если } x < 0 \end{cases} \quad comp(x) = \begin{cases} x, & \text{если } x \geq 0 \\ 2^n - |x|, & \text{если } x < 0. \end{cases}$$

Например, при $n = 8$

$$inv(+1) = comp(+1) = 1 = 00000001_2;$$

$$inv(-1) = 2^8 - 1 - |-1| = 255 - 1 = 254 = 11111110_2;$$

$$comp(-1) = 2^8 - |-1| = 256 - 1 = 255 = 11111111_2;$$

$$inv(-127) = 2^8 - 1 - |-127| = 255 - 127 = 128 = 10000000_2;$$

$$comp(-127) = 2^8 - |-127| = 256 - 127 = 129 = 10000001_2;$$

$$comp(-128) = 2^8 - |-128| = 256 - 128 = 128 = 10000000_2;$$

В обратном коде число -128 не представимо.

Из трех рассмотренных кодов с точки зрения эффективности выполнения арифметических операций чаще всего применяются последние два, причем наиболее эффективный из них – дополнительный код. Кодирование чисел в обратный и дополнительный код позволяет существенно упростить устройство АЛУ путем замены некоторых арифметических операций сложением.

1.3.2 Выполнение арифметических операций над целыми числами

Сложение и вычитание беззнаковых чисел происходит по обычным для позиционных систем счисления алгоритмам, рассмотренным ранее в п.1.2.3. Ситуации, когда уменьшаемое меньше вычитаемого или когда результат суммы не укладывается в n разрядов, считаются ошибочными и должны отслеживаться устройством компьютера.

Сложение в обратном коде происходит следующим образом: по обычному алгоритму складываются все разряды, включая знаковый. Результат такого сложения для n -разрядных двоичных чисел имеет длину $n+1$ (самый левый разряд результата равен единице, если был перенос при сложении старших разрядов операндов, иначе – нулю). Значение левого $n+1$ -го разряда добавляется к младшему разряду результата. Получаем n -разрядное двоичное число, которое и будет суммой двух чисел в обратном

коде. Некоторые возможные случаи сложения операндов представлены в табл. 1.3.

Вычитание чисел в обратном коде $x - y$ сводится к сложению $x + (-y)$.

Таблица 1.3 – Возможные случаи сложения двух знаковых двоичных чисел (A+B) с применением обратного кода

Возможные случаи	В десятичном виде	В двоичном виде
A>0 B>0	$\begin{array}{r} + 5 \\ + 10 \\ \hline 15 \end{array}$	$\begin{array}{r} + 0\ 0000101 \\ + 0\ 0001010 \\ \hline 0\ 0001111 = 15_{10} \end{array}$
A>0 B<0 B >A	$\begin{array}{r} + 5 \\ + -15 \\ \hline -10 \end{array}$	$\begin{array}{r} + 0\ 0000101 \\ + 1\ 1110000 \\ \hline 1\ 1110101 = -10_{10} \end{array}$
A>0 B<0 B <A	$\begin{array}{r} + 15 \\ + -5 \\ \hline 10 \end{array}$	$\begin{array}{r} + 0\ 0001111 \\ + 1\ 1111010 \\ \hline 0\ 0001001 \\ 1 \text{ (перенос)} \\ \hline 0\ 0001010 = 10_{10} \end{array}$
A<0 B<0	$\begin{array}{r} + -5 \\ + -15 \\ \hline -15 \end{array}$	$\begin{array}{r} + 1\ 1111010 \\ + 1\ 1110101 \\ \hline 1\ 1101111 \\ 1 \text{ (перенос)} \\ \hline 1\ 1110000 = -15_{10} \end{array}$

При сложении двух двоичных чисел может произойти ситуация когда разряды не помещаются в выделенную для них область памяти. Такая ситуация называется *переполнением разрядной сетки формата числа*. Для обнаружения переполнения и оповещения о нем используются специальные средства. Так, в некоторых компьютерах при делении на ноль вычисления прекращаются (фатальная ошибка), а при переполнении мантиссы устанавливается признак переполнения в так называемом регистре флагов и вычисления продолжают.

Убедиться в том, что произошло переполнение можно по знаковому разряду. Переполнение разрядной сетки имеет место в случае, когда знак суммы не совпадает со знаками обоих операндов, при условии, что складываются операнды с одинаковыми знаками. В табл. 1.4 приведены примеры подобного сложения.

Сложение чисел с разными знаками не приводит к переполнению.

Таблица 1.4 – Ситуации переполнения при сложении двух знаковых двоичных чисел (A+B)

Возможные случаи	В десятичном виде	В двоичном виде
$A > 0, B > 0,$ $ A + B \geq 2^{n-1}$	$\begin{array}{r} + 85 \\ + 73 \\ \hline 158 \end{array}$	$\begin{array}{r} + 0\ 1010101 \\ + 0\ 1001001 \\ \hline 1\ 0011110 \end{array}$ Результат некорректен
$A < 0, B < 0,$ $ A + B \geq 2^{n-1}$	$\begin{array}{r} + -85 \\ + -73 \\ \hline -158 \end{array}$	$\begin{array}{r} + 1\ 0101010 \\ + 1\ 0110110 \\ \hline 0\ 1100000 \\ 1 \oplus \\ \hline 0\ 1100001 \end{array}$ Результат некорректен

В дополнительном коде сложение происходит так: по обычному алгоритму складываются все разряды, включая знаковый; единица переноса в $n + 1$ -й разряд отбрасывается. Например, при $n = 4$:

$$5_{10} + (-3_{10}) = 0101_2 + 1101_2 = (1)0010_2 \Rightarrow 0010_2 = 2_{10}.$$

При вычитании тоже действует обычный алгоритм, причем, если уменьшаемое меньше вычитаемого, к двоичному коду уменьшаемого слева приписывается единица и только после этого производится вычитание. Например, при $n = 4$:

$$3_{10} - 7_{10} = 0011_2 - 0111_2 \Rightarrow 1\ 0011_2 - 0111_2 = 1100_2 = -4_{10}.$$

Заметим, что на преобразование числа в обратный код компьютер затрачивает меньше времени, чем для преобразования его в дополнительный код, так как последнее состоит из двух шагов – образования обратного кода и прибавления единицы к его младшему разряду. Время выполнения сложения для дополнительных кодов чисел меньше, чем для их обратных кодов, потому что в таком сложении нет переноса единицы из знакового разряда в младший разряд результата. Здесь единицы просто отбрасываются.

Умножение, в сравнении с рассмотренными ранее операциями сложения и вычитания, является более сложной операцией. При умножении двух целых беззнаковых чисел размером в n бит максимальный результат, получаемый процессором, может иметь длину $2n$ бит. В большинстве компьютеров умножение производится как последовательность сложений и сдвигов. Для этого в АЛУ имеется регистр, называемый *накапливающим*

сумматором, который до начала выполнения операции содержит число ноль. В процессе выполнения операции в нем поочередно размещаются множимое и результаты промежуточных сложений, а по завершении операции – окончательный результат. Другой регистр АЛУ, участвующий в выполнении этой операции, вначале содержит множитель. Затем по мере выполнения сложений содержащееся в нем число уменьшается, пока не достигнет нулевого значения. Приведем пример умножения двух положительных чисел при $n=4$:

$$\begin{array}{r}
 \times 1001_2 \\
 \underline{1011_2} \\
 + 1001_2 \\
 + 1001_2 \\
 \underline{1001_2} \\
 1100011_2
 \end{array}$$

$1001_2 = 9, \quad 1011_2 = 11, \quad 9 \times 11 = 99,$
 $1100011_2 = 2^7 + 2^6 + 2^1 + 2^0 = 64 + 32 + 2 + 1 = 99.$

Операция деления над множеством целых чисел со знаком не определена, поскольку в общем случае ее результатом будет вещественное число. Однако допустимыми являются операции целочисленного деления и нахождения остатка от целочисленного деления. Обычно они реализуются путем многократного прибавления к делимому дополнительного кода делителя.

1.3.3 Числа с плавающей точкой

Вещественными числами в компьютерной технике называются числа, имеющие дробную часть. В математических вычислениях система вещественных чисел предполагается бесконечно непрерывной, не имеющей ограничений на диапазон и точность их представления. Понятно, что реализовать такую систему в технических устройствах не представляется возможным из-за фиксированных размеров ячеек памяти. Таким образом, система машинных чисел является конечной и дискретной, образуя подмножество системы вещественных чисел.

Вещественные числа могут принимать значения из широкого диапазона и могут быть как очень маленькими, так и очень большими. Для их записи принято использовать единый вид представления с плавающей точкой.

Число с плавающей точкой – это формат представления числа, при котором оно записывается в виде следующего произведения:

$$N = \pm M \cdot q^{\pm p}.$$

где N – значение вещественного числа; M – мантисса числа; q – основание системы счисления; p – порядок числа.

Мантисса (дробное число со знаком) и *порядок* (целое число со знаком) изображаются в системе счисления с основанием q , а само основание – в десятичной системе. Знак числа совпадает со знаком мантиссы. Порядок определяет положение точки в числе.

Максимальная точность представления вещественного числа в компьютере, при фиксированном количестве разрядов, отведённых под мантиссу, обеспечивается тогда, когда точка расположена в мантиссе перед первой значащей цифрой. В этом случае возможна запись максимального количества значащих цифр числа. Такое представление вещественных чисел называется нормализованным.

Нормализация – это представление мантиссы в диапазоне от q^{-1} до 1, где q – основание системы счисления. В десятичной системе счисления мантисса нормализованного числа лежит в диапазоне $0.1 \leq M_{10} < 1$. В двоичной системе счисления $q=2$ и мантисса принимает значения от 2^{-1} до 1, т.е. имеет вид правильной дроби (с единицей после запятой и нулем в целой части числа).

Рассмотрим примеры нормализованного представления вещественных чисел:

десятичная система	двоичная система
$6895.35 = 0.689535 \cdot 10^4$;	$-110.01 = -0.11001 \cdot 2^{11}$ (порядок $11_2 = 3_{10}$)
$-0.000786 = -0.786 \cdot 10^{-3}$;	$0.000011 = 0.11 \cdot 2^{-100}$ (порядок $-100_2 = -4_{10}$)

Вещественные числа в компьютерах разных типов записываются по-разному. Тем не менее, практически все компьютеры поддерживают международный стандарт IEEE 754, регламентирующий формат записи чисел с плавающей точкой. Этот стандарт определяет два базовых формата – 32- и 64-битовый (рис.1.4). Первый вариант называется представлением с обычной точностью, второй – представлением с двойной точностью⁴.

Мантисса, называемая также «дробью» F (Fraction), представлена в прямом коде. S – бит знака. Порядок E (Exponent) дается в смещенной форме и равен истинному порядку, увеличенному на значение смещения. Если для задания порядка выделено n разрядов, то к истинному значению порядка, прибавляют смещение равное $2^{n-1} - 1$. Значение смещения для базовых форматов равно 127 (8 бит) и 1023 (10 бит). Задание порядка в форме со смещением упрощает операцию сравнения чисел с плавающей точкой, превращая ее в операцию сравнения целых чисел. Смещенный порядок называется также характеристикой, ее можно считать целым положительным и беззнаковым числом.

⁴ Кроме двух основных, стандарт регламентирует также два расширенных формата, в которых зарезервированы дополнительные биты для порядка и мантиссы.

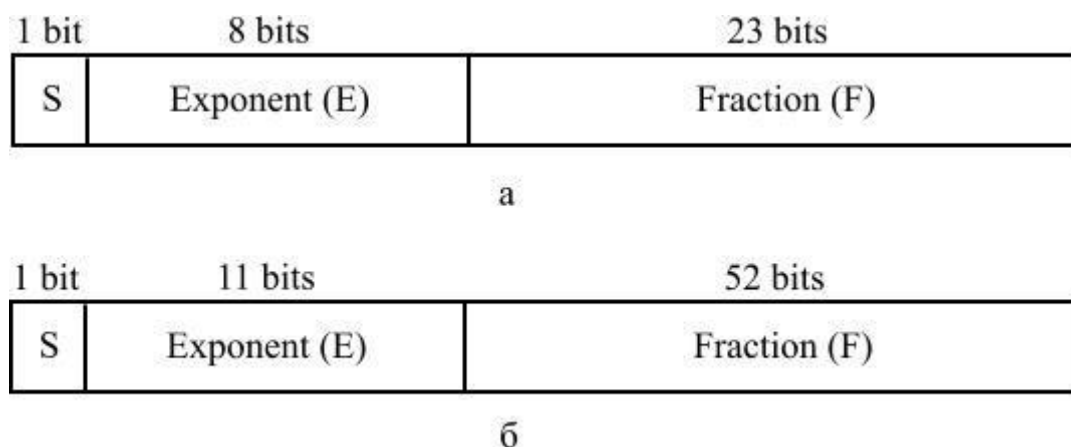


Рис.1.4 - Форматы чисел с плавающей точкой в стандарте IEEE 754:
32-битовый формат (а); 64-битовый формат (б)

Отметим, что в стандарте старший бит нормализованной мантиссы содержит единицу и расположен слева от точки, так называемый бит единицы. В форматах с обычной и двойной точностью данный бит при передачах чисел и хранении их в памяти не фигурирует. Он является скрытым или неявным. Таким образом, фактически мантисса принадлежит интервалу $1 \leq M < 2$.

Рассмотрим пример. Пусть задано вещественное двоичное число в 32-битовом формате с плавающей точкой: 11000000 10110000 00000000 00000000₂. Найдем десятичное значение числа. $S=1$, $E=10000001$ (8 бит), $F=0110000 00000000 00000000$ (23 бита). В нормализованной форме мантисса нормирована с неявным битом единицы в виде $1.F$, т.е. фактически $M=1.F=1,0110000 00000000 00000000 = 1 + 2^{-2} + 2^{-3} = 1.375_{10}$. В примере $S=1$, значит число отрицательное, т.е. -1.375_{10} . Порядок числа в памяти компьютера хранится в прямом коде со смещением. Найдем его: $E = 10000001_2 = 129_{10}$. Найдем реальный порядок, от порядка со смещением отнимем величину смещения 127_{10} : $P=E-127 = 129-127 = 2_{10}$. Теперь можно найти число: $-1.375 \cdot 2^2 = -5.5_{10}$.

В общем виде нормализованные вещественные числа записываются в следующей форме: $(-1)^S \cdot 1.F \cdot 2^{(E-\text{смещение})}$.

Заметим, что в нормализованной форме неявный бит единицы в мантиссе не может быть равен 0. Для представления числа 0, а также других чисел с абсолютными значениями близкими к нулю, была разработана денормализованная форма. Денормализованная форма имеет схожую запись с нормализованной формой, однако у нее фактический порядок всегда равен наименьшему возможному значению порядка для нормализованных чисел (-126 для обычной точности и -1022 для двойной), а мантисса $M=0.F$.

Таким образом, число 0 в 32-битовом формате может быть представлено с $E=0$ и $F=0$, т.е. $0.0 \cdot 2^{-126} = 0$. Причем, имеется два представления нуля $+0$ (когда $s = 0$) и -0 (когда $s = 1$).

Аналогично в денормализованной форме можно представлять числа близкие к нулю. Например, если $S=1$, $E=0$, и $F=0110000\ 00000000\ 00000000_2$. Мантисса $M = 0,011 = 2^{-2} + 2^{-3} = 0.375_{10}$. Так как $S=1$, то это отрицательное число. При $E=0$, фактический порядок равен -126. Таким образом, получаем число $-0.375 \cdot 2^{-126} = -4.4 \cdot 10^{-39}$.

Если смещенный порядок равен максимально возможному значению ($E=255$ для обычной точности и $E=2047$ для двойной), то число представляет собой специальное значение INF (положительные и отрицательные бесконечности) или NaN (англ. Not a Number – не число). NaN обозначает значение, которое не может быть представлено в виде вещественного числа (например, результат деления $0/0$).

Диапазон чисел с плавающей точкой зависит от количества разрядов порядка и мантиссы. Чем больше разрядов отводится для записи мантиссы, тем выше точность представления числа. Чем больше разрядов занимает порядок, тем шире диапазон от меньшего, отличного от нуля числа, до наибольшего, представленного в компьютере при заданном формате.

Определим диапазон нормализованных вещественных чисел в 32-битовом формате. Максимальное значение мантиссы $M_{\max}=2$, а минимальное $M_{\min}=1$ для положительных чисел и $M_{\max}=-1$, а минимальное $M_{\min}=-2$ для отрицательных чисел. Максимальное значение порядка $P_{\max}=254$, а минимальное $P_{\min}=1$. Смещение равно 127. Определим диапазон положительных нормализованных вещественных чисел:

$$\text{от } N_{\min}=M_{\min} \cdot 2^{(1-127)} \approx 1,17 \cdot 10^{-38} \text{ до } N_{\max}=M_{\max} \cdot 2^{(254-127)} \approx 3,4 \cdot 10^{38}$$

Диапазоны абсолютных значений нормализованных и денормализованных чисел с плавающей точкой обычной и двойной точностей приведены в табл.1.5. На рис. 1.5 видно, что в диапазоне представимы не все вещественные числа.

Таблица 1.5 – Диапазоны абсолютных значений нормализованных и денормализованных чисел в формате с плавающей точкой

Точность	Нормализованное число $N_{\min}$	Нормализованное число $N_{\max}$
обычная	$1,17549435 \cdot 10^{-38}$	$3,4028235 \cdot 10^{38}$
двойная	$2,2250738585072014 \cdot 10^{-308}$	$1,7976931348623157 \cdot 10^{308}$
	Денормализованное число $D_{\min}$	Денормализованное число $D_{\max}$
обычная	$1,4 \cdot 10^{-45}$	$1,1754942 \cdot 10^{-38}$
двойная	$4,9 \cdot 10^{-324}$	$4,4501477170144023 \cdot 10^{-308}$



Рис.1.5 – Диапазон представления вещественных чисел

Рассмотрим, как осуществляются арифметические операции над нормализованными вещественными числами. При сложении и вычитании сначала производится подготовительная операция, называемая выравниванием порядков. В процессе выравнивания мантисса числа с меньшим порядком сдвигается в своем регистре вправо на количество разрядов, равное разности порядков операндов. После каждого сдвига порядок увеличивается на единицу. В результате выравнивания порядков одноименные разряды чисел оказываются расположенными в соответствующих разрядах обоих регистров, после чего мантиссы складываются или вычитаются.

В случае необходимости полученный результат нормализуется путем сдвига мантиссы результата влево. После каждого сдвига влево порядок результата уменьшается на единицу.

Сложим два двоичных нормализованных числа $0.10111 \cdot 2^{-1}$ и $0.11011 \cdot 2^{10}$. Разность порядков слагаемых равна трем, поэтому перед сложением мантисса первого числа сдвигается на три разряда вправо: $0.00010111 \cdot 2^{10}$. Теперь числа можно сложить.

$$\begin{array}{r}
 0.00010111 \cdot 2^{10} \\
 + 0.11011 \cdot 2^{10} \\
 \hline
 0.11101111 \cdot 2^{10}
 \end{array}$$

При умножении двух нормализованных чисел их порядки складываются, а мантиссы перемножаются. При делении двух нормализованных чисел из порядка делимого вычитается порядок делителя, а мантисса делимого делится на мантиссу делителя. Затем в случае необходимости полученный результат нормализуется.

Выполнить умножение двоичных нормализованных чисел: $0.11101 \cdot 2^{101}$ и $0.1001 \cdot 2^{11}$:

$$(0.11101 \cdot 2^{101}) \cdot (0.1001 \cdot 2^{11}) = (0.11101 \cdot 0.1001) \cdot 2^{(101+11)} = 0.100000101 \cdot 2^{1000}$$

1.4 Команды компьютера

Команда – это описание элементарной операции, которую должен выполнить компьютер. Обычно программисты работают с операторами конкретного языка программирования. Однако следует понимать, в какие фактические команды компьютера транслируются операторы языка программирования. Команды, представленные в форме, в которой они непосредственно используются устройством управления компьютера, называются *машинными командами*, а о программах, написанных машинными командами, говорят, что они написаны в *машинном коде*.

Машинные команды хранятся в ячейках памяти в двоичном коде и обычно имеют размер от двух до четырех байтов. Адрес команды определяется как адрес ее первого байта.

Рассмотрим общую структуру команды. Она содержит код команды, определяющий ее тип – операцию, выполняемую командой, а также ноль или более операндов, которые могут быть адресами данных, над которыми должна быть выполнена операция, или значениями. В явной и неявной форме команда содержит также информацию об адресе, по которому помещается результат операции, и об адресе следующей команды. Таким образом, в общем случае команда состоит из: операционной части (содержит код операции) и адресной части (содержит адресную информацию о местонахождении обрабатываемых данных и месте хранения результатов).

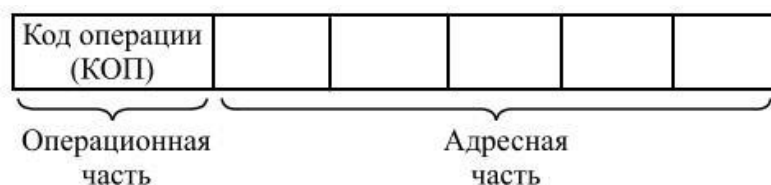


Рис.1.6 – Обобщенная структура команды

В зависимости от количества операндов в адресной части, команды бывают: одноадресные, двухадресные, трехадресные, переменнаядресные.

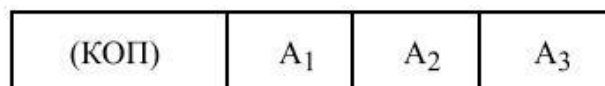


Рис.1.7 – Упрощенная структура трехадресной команды

В трехадресной команде (рис. 1.7) адреса A₁ и A₂ указывают ячейки памяти, в которых расположены операнды, а A₃ определяет ячейку, в которую помещается результат операции.

При условии, что результат операции всегда помещается на место одного из операндов, получим двухадресную команду. В одноадресной команде один из операндов указывается адресом в команде, а в качестве

второго используется содержимое регистра процессора, называемого *аккумулятором (сумматором)*. Результат операции записывается в тот же регистр.

Команды программы выполняются последовательно, начиная с первой. В начале каждого цикла процессор выбирает из памяти команду. Адрес ячейки, из которой нужно извлечь очередную команду, хранится в *регистре – счетчике команд*. После извлечения каждой команды процессор увеличивает значение счетчика команд на длину текущей команды так, чтобы получить адрес следующей команды. Таким образом, команды выполняются в порядке возрастания номеров ячеек памяти, в которых они хранятся. Извлеченная команда передается в устройство управления, где загружается в *регистр команд*. Устройство управления расшифровывает адресное поле команды и по его сигналам операнды считываются из памяти и записываются в АЛУ в специальные *регистры операндов*. На следующем этапе УУ расшифровывает код операции и выдает в АЛУ сигнал выполнить соответствующую операцию над данными. Результат операции либо остается в процессоре, либо отправляется в память, если в команде был указан адрес результата. Далее все предыдущие этапы повторяются до достижения команды, содержащей код операции, останавливающей вычисления.

Компьютеры имеют команды трех разных типов:

–Вычисления: арифметические операции, такие как сложение, вычитание, умножение, деление для целых и вещественных чисел, операции сравнения, побитовые операции. Обычно такие операции выполняются над операндами, расположенными в регистрах.

–Загрузки и хранения: загрузка данных из оперативной памяти в регистры и обратно, а также обеспечение обмена данными с устройствами ввода/вывода.

–Управления: переключение выполнения к другой точке программного кода в зависимости от выполнения некоторого условия или безусловного перехода. Так, команда может задавать изменение последовательности выполнения команд. Например, если процессор извлекает команду, которая указывает, что следующей по очереди должна быть исполнена команда из определенной ячейки, то процессор устанавливает значение счетчика команд равным этой ячейке и в следующем цикле выборки из нее будет извлечена следующая команда.

1.5 Классификация языков программирования

По мере развития компьютерной индустрии стало ясно, что программирование, которое производится непосредственно на машинных кодах дело медленное. Представление программы в машинных командах не является удобной формой для человека. Ситуация осложнялась еще и тем, что каждая вычислительная машина понимает лишь свой машинный язык.

Для упрощения этой задачи стали появляться языки программирования низкого уровня, которые позволяли задавать машинные команды в более понятном для человека виде. Так вместо последовательности чисел, которую компьютер может понимать напрямую, программисты для обозначения элементарных операций компьютера стали использовать символьные обозначения. Стремление человека оперировать словами и не цифрами привело к появлению *ассемблеров* (англ. Assembly Language). Это языки, в которых вместо численного обозначения команд и областей памяти используются словесно-буквенные.

При этом появилась проблема: компьютер не в состоянии понять наборы букв. Необходим какой-нибудь переводчик на его родной машинный язык. Поэтому, начиная со времен ассемблеров, под каждый язык программирования создаются *трансляторы* (англ. translator) – специальные программы, преобразующие программный код с языка программирования в машинный код. Для ассемблерных языков выполняется соответствие «один в один» – одна команда ассемблера, как правило, транслируется в одну машинную команду компьютера. На рис. 1.8 показана схема трансляции команд ассемблерного языка в машинные коды. Во фрагменте программы осуществляется сложение операндов R1 и R2, сумма сохраняется в R5.



Рис.1.8 – Схема трансляции команд ассемблерного языка в машинные коды

Языки ассемблерного типа являются *языками программирования низкого уровня*. Они ориентированы на конкретный тип процессора и учитывают его особенности, поэтому для переноса программы на ассемблере на другую аппаратную платформу ее нужно почти полностью переписать. С помощью языков низкого уровня создаются эффективные и компактные программы, поскольку разработчик получает доступ ко всем возможностям процессора. Языки низкого уровня, как правило, используют для написания небольших системных программ, драйверов устройств, программирования специализированных микропроцессоров, когда важнейшими требованиями являются компактность, быстродействие и возможность прямого доступа к аппаратным ресурсам.

После ассемблеров наступил рассвет *языков программирования высокого уровня* (англ. High Level Language). Для этих языков потребовалось разрабатывать более сложные трансляторы, так как языки высокого уровня куда больше удобны для человека, чем для вычислительной машины. В отличие от ассемблеров, которые остаются привязанными к конкретной

архитектуре компьютера, языки высокого уровня являются переносимыми (независимыми от платформы). Написав один раз программу на языке высокого уровня, программист мог выполнить ее на любом компьютере. Один оператор языка высокого уровня может выполнять целый ряд сложных действий. Так, ранее рассмотренный фрагмент программы на ассемблере будет иметь следующий вид на языке высокого уровня: $R5 = R1 + R2$. Разрабатывать программы на таких языках гораздо проще и ошибок допускается меньше. Значительно сокращается время разработки программы, что особенно важно при работе над большими программными проектами.

К языкам программирования высокого уровня относятся такие языки, как: Fortran, Pascal, Delphi, C++, C#, Java, Python, PHP, Ruby, Perl. Недостатком языков высокого уровня является большой размер программ по сравнению с программами на языке низкого уровня. Поэтому в основном языки высокого уровня используются для разработок программного обеспечения компьютеров и устройств, которые имеют большой объем памяти. А разные подвиды ассемблера применяются для программирования других устройств, где критичным является размер программы.

1.6 Этапы создания и подготовки программы к выполнению

Разработка программы начинается с постановки задачи. На этом этапе формулируется цель решения задачи и подробно описывается ее содержание. Проводится анализ условий, при которых решается поставленная задача и выявляется область определения входных параметров. В дальнейшем это позволяет выполнить формализованное описание задачи и построить ее математическую модель.

Следующим важным этапом является *проектирование программы*. На данном этапе обычно осуществляется выбор и обоснование метода решения, при этом учитываются влияния различных факторов и условий на конечный результат, в том числе на точность вычислений, время решения задачи на компьютере, требуемый объем памяти и прочее. Составляется алгоритм решения задачи, в соответствии с выбранным методом решения. Процесс обработки данных разбивается на отдельные относительно самостоятельные блоки, определяется последовательность выполнения этих блоков. Модель решения детализируется и разбивается на подпрограммы, определяется структура программы и способ хранения информации. Обычно алгоритм несколько раз уточняется, перед тем как он примет свой окончательный вид, удобный для написания программы.

Запись алгоритма на языке программирования принято называть *кодированием*. Программа может быть написана на одном языке программирования или иногда разные части программы пишут на разных языках. Чаще для этого используют языки высокого уровня. Современные системы программирования позволяют ускорить процесс разработки программы, автоматически создавая часть ее текста, однако творческая работа по-прежнему лежит на программисте.

Этап устранения синтаксических⁵ и семантических⁶ ошибок в программе называется *отладкой* программы. В процессе трансляции программы выявляются недопустимые конструкции и символы для данного языка программирования. Компьютер выдает сообщение об ошибках в форме, соответствующей этому языку.

Убедиться в том, что полностью отлаженная программа дает правильный результат, позволяет этап *тестирования*. Для этого разрабатывается система тестов – специально подобранных контрольных примеров с такими наборами параметров, для которых решение задачи известно. Тестирование должно охватывать все возможные ветвления в программе, т.е. проверять все ее инструкции, и включать такие исходные данные, для которых решение невозможно.

Разработанная программа поставляется заказчику в виде готовой к исполнению машинной программы. К ней прилагается сопроводительная документация, включающая «Руководство разработчика программы» и «Руководство пользователя». Первый документ дает инструкции по исправлению ошибок во время эксплуатации программы и ее дальнейшему развитию, а в другом документе объясняется, как использовать программу и работать с ней пользователю.

Все приведенные этапы тесно связаны между собой. Например, тестирование и анализ результатов может привести к необходимости внесения изменений в программу, алгоритм, метод решения или даже в постановку задачи.

В настоящее время существуют множество языков программирования: как достаточно универсальных, так и очень специфических. Многие программисты старались и стараются придумать свой язык обладающий теми или иными преимуществами. Можно лишь условно разделить языки по определенным критериям. Например, по применяемой технологии программирования. Среди прочего выделяют технологии структурного и объектно-ориентированного программирования.

В *технологии структурного программирования* (некоторые представители: Fortran, Pascal, C) от входных данных полностью зависит последовательность выполнения команд. Структурная программа предписывает, как достичь результата, пошагово описывая действия. Поэтому такое программирование является достаточно легким для понимания. Именно, в структурном программировании в свое время возникла концепция локализации части кода в так называемые подпрограммы (функции, методы), с последующим их вызовом из разных мест основной программы. При вызове в подпрограмму могут передаваться

⁵ Синтаксические ошибки – это ошибки в записи конструкций языка программирования (чисел, переменных, функций, выражений, операторов, меток, подпрограмм).

⁶ Семантические ошибки – это ошибки, связанные с неправильным содержанием действий и использованием недопустимых значений величин.

какие-либо данные в виде аргументов; а подпрограмма, в свою очередь, может возвращать в главную программу результат – данные, полученные в ходе ее выполнения.

Технология объектно-ориентированного программирования (некоторые представители: C++, Java, Python) призвана решать более сложные и объемные задачи по сравнению со структурным программированием. Особое внимание уделяется данным, которые представляются в программе в виде объектов. Объекты взаимодействуют между собой с помощью механизма передачи сообщений. Задача программиста – реализовать такие объекты, при взаимодействии которых можно будет получать желаемый результат.

Прежде чем программа, написанная на языке высокого уровня, сможет выполняться, ее необходимо перевести на машинный язык, связать с другими программами на машинном языке, от которых она зависит, и загрузить в основную память. Языки высокого уровня, в зависимости от особенностей преобразования их кода, можно разделить на компилируемые, компилируемые в байт-код и интерпретируемые.

Компиляторы (англ. compiler) – это программы, которые преобразуют исходные тексты программ, написанные на языке программирования высокого уровня, в машинный или *нативный код* (англ. native code), который можно сохранить и затем использовать уже без компилятора (примером являются исполняемые файлы с расширением *.exe). Недостатком компилируемых языков является то, что сгенерированный нативный код нельзя перенести на другую платформу. Если программу нужно запустить на другом типе процессора, ее нужно перекомпилировать так, чтобы исполняемый код включал команды этого нового процессора. То есть, нужен другой компилятор.

Рассмотрим, как осуществляется сборка программ, написанных на компилируемых языках. Исходный код программы, созданный и отредактированный с помощью текстового редактора, компилируется в так называемый объектный файл, который далее обрабатывается еще одной программой – *компоновщиком* (редактором связей), которая компоует (собирает) полный код программы и загружает его в оперативную память, или записывает на диск в виде готового исполняемого файла (рис.1.9).



Рис.1.9 – Схема создания исполняемого файла

Как видно, прежде чем получить из исходного кода исполняемый файл, необходимо скомпилировать его в объектный файл. Формат объектных файлов единый для программ, написанных на разных языках программирования. Это позволяет компоновщику связывать в исполняемый

файл отдельные программы, написанные на разных языках. Кроме того, если компилируемый проект состоит из многих исходных файлов, то при изменении одного из них нет необходимости перекомпилировать весь исходный код, достаточно откомпилировать один файл в один объектный файл и запустить стадию компоновки. Это позволяет существенно экономить время создания исполняемого файла.

Интерпретаторы (англ. interpreter) выполняют аналогичную компиляторам функцию, но делают это построчно всякий раз во время исполнения программы, по мере поступления её исходного кода на вход интерпретатора. Интерпретатор читает исходный текст программы, строчку за строчкой, команду за командой, анализирует каждую команду и исполняет ее. Чтобы исполнить программу, написанную на интерпретируемом языке, на другом типе процессора перекомпилировать уже ничего не нужно, просто нужно иметь интерпретатор, работающий на этом новом процессоре.

У обоих подходов есть свои преимущества и недостатки. Интерпретируемая программа не нуждается в перекомпиляции при внесении изменений или при переносе на другую платформу. Это, безусловно, преимущество по сравнению с компилируемой программой. Однако есть и очень существенный недостаток: в отличие от скомпилированной программы, где исполняемый код состоит из команд процессора и непосредственно процессором же и исполняется, в интерпретируемой программе интерпретатор должен проанализировать каждую команду и исполнить ее. Таким образом, чтобы исполнить каждую команду, надо выполнить много всяких дополнительных действий по анализу текста команды, что приводит к значительному замедлению работы. В компилируемой программе эти действия выполняются однократно в процессе компиляции и, соответственно, при исполнении программы не нужны. Поэтому интерпретируемая программа может исполняться в несколько десятков, а иногда и сотен раз медленнее аналогичной компилируемой программы.

Разницу двух подходов иллюстрирует рис.1.10. Как видно из рисунка, у интерпретатора два источника ввода – исходная программа и входные данные, а на вход компилятору подается только исходный код, входные данные вводятся уже при выполнении исполняемого файла.

Компиляция в байт-код – это подход, при котором программа преобразуется в промежуточный двоичный код, похожий по структуре на машинный код реальных процессоров, поэтому его можно быстро и эффективно интерпретировать во время исполнения практически на любом из существующих процессоров с небольшой потерей производительности. Можно рассматривать данный подход, как попытку совместить компиляцию и интерпретацию и избавиться от недостатков обоих. Программа сначала компилируется, но в результате получается не исполняемый код, состоящий из команд процессора, а так называемый *байт-код* (англ. byte-code). Байт-код в дальнейшем интерпретируется. Основная идея байт-кода заключается в том, что он оптимизирован с точки зрения интерпретации. То есть, это не

исходный текст программы, а некий вид записи инструкций программы, который можно интерпретировать во много раз быстрее, чем исходный текст. В итоге это приводит к тому, что скорость исполнения интерпретируемого байт-кода сравнима со скоростью исполнения скомпилированной программы.

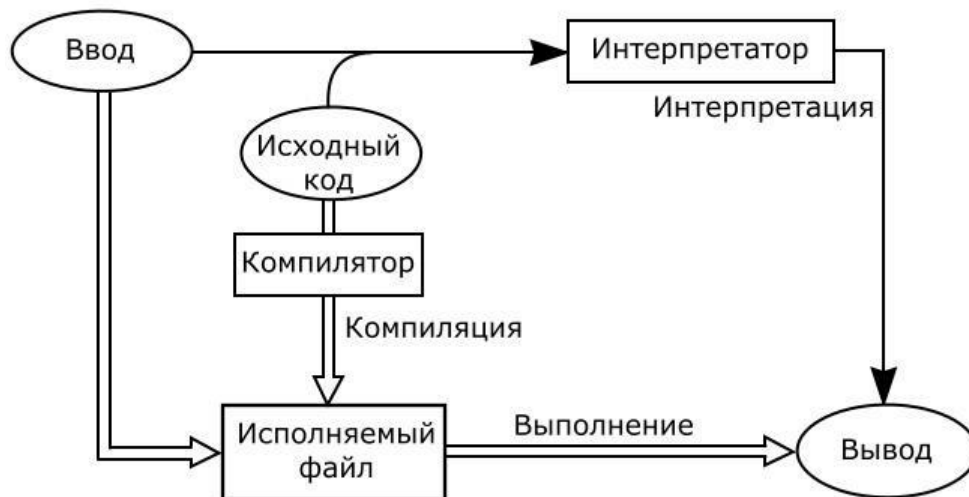


Рис.1.10 – Компиляция и интерпретация

Рассмотрим процесс сборки программ, написанных на языках, компилируемых в байт-код, на примере языка Java. Компиляторы Java компилируют программу не в машинный код, а в байт-код, который также можно считать языком машинных команд, поскольку в качестве его интерпретатора используется так называемая *виртуальная машина Java* (англ. Java Virtual Machine – JVM). Байт-код не является машинным кодом какого-либо компьютера, т.е. машинным кодом в прямом смысле слова, и может переноситься на различные компьютерные архитектуры без перекомпиляции, так как интерпретатор виртуальной машины пишется для каждой целевой платформы (рис.1.11). Скорость интерпретации байт-кодов ненамного медленнее скорости исполнения скомпилированной программы.

Возможен также вариант компиляции программы на байт-коде в машинный код непосредственно перед исполнением, так называемая «Just-In-Time компиляция» (JIT), по названию подобного компилятора для Java. Использование динамических JIT-компиляторов, называемых также джитерами, увеличивает производительность приложений. В этом случае не виртуальная машина исполняет байт-код, а JIT-компилятор преобразует его в «родной» для данной машины код (рис.1.12). Таким образом, с одной стороны, повышается производительность скомпилированного кода, а с другой – сохраняется его переносимость. Обычно, как показано на рисунке, наряду с JIT-компиляцией (джитингом) остается и возможность интерпретации байт-кода. JIT-компиляция обычно имеет место при первом использовании модуля (метода или всего класса), так что она будет нужна только для кода, фактически используемого в данном сеансе выполнения.

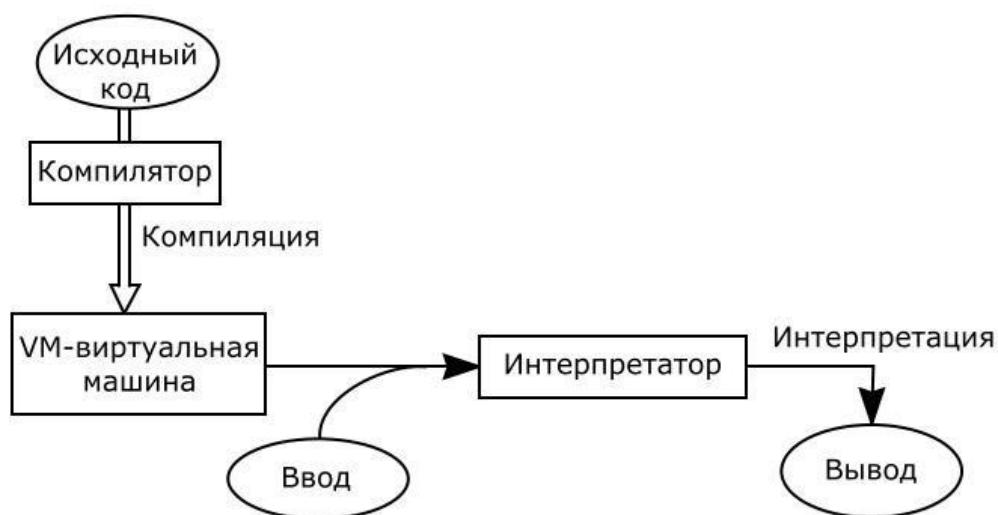


Рис.1.11 – Схема компиляции и интерпретации байт кода

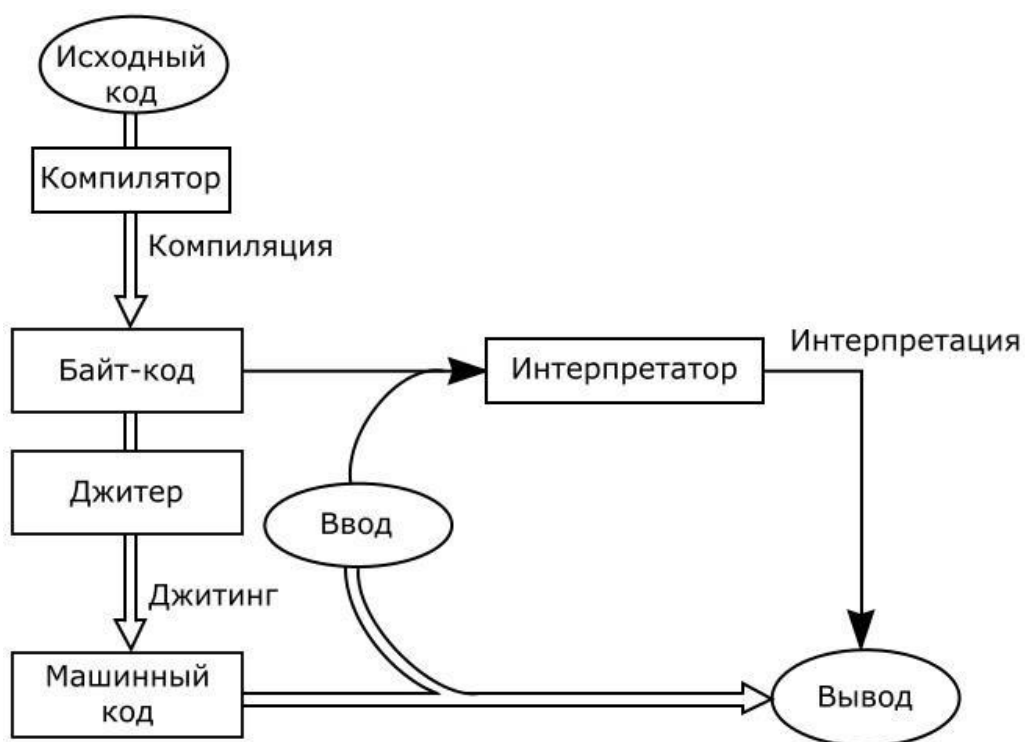


Рис.1.12 – Схема компиляции байт-кода с помощью JIT-компилятора

Все перечисленные выше действия требуют наличия специальных программных средств: текстового редактора, компилятора и/или интерпретатора, компоновщика, загрузчика и отладчика. Совокупность этих программных средств образует *систему программирования* или *интегрированную среду разработки* (англ. Integrated Development Environment, IDE). Кроме них в состав IDE входит библиотека стандартных подпрограмм, которые можно использовать во время создания программы.

1.7 Алгоритмы и основные алгоритмические структуры

С давних пор человеку приходится создавать описания последовательностей действий, требуемых для достижения некоторой поставленной цели. Например, в математике для решения типовых задач мы используем определенные правила, описывающие последовательности действий: правила сложения дробных чисел, решения квадратных уравнений и т.д. Предписание, определяющее порядок выполнения действий над данными с целью получения искомых результатов, и есть алгоритм.

*Алгоритм*⁸ (англ. Algorithm) – заранее заданное понятное и точное предписание возможному исполнителю совершить определенную последовательность действий для получения решения задачи за конечное число шагов.

Формальная система, предназначенная для записи алгоритмов, называется *алгоритмическим языком*, сам текст алгоритма – *программой*, а процесс его создания – *программированием*.

Наука программирование (англ. Computer Science) занимается исследованием свойств алгоритмов и разработкой методов построения программ.

1.7.1 Свойства и формы представления алгоритма

Рассмотрим пример алгоритма, а потом перечислим основные свойства, которыми должен обладать любой алгоритм.

Представим в виде алгоритма решение задачи нахождения наименьшего простого⁹ делителя натурального числа N , большего единицы.

Алгоритм:

Шаг 1: Положить целое число i равным двум и перейти на шаг 2.

Шаг 2: Если N делится нацело на i , то завершить работу алгоритма, выдав в качестве результата i ; иначе перейти на шаг 3.

Шаг 3: Увеличить значение i на единицу и перейти на шаг 2.

Для того чтобы понять этот алгоритм, надо выступить в роли исполнителя команд, выполняя вручную указанную в нем последовательность действий для некоторых небольших значений N . Будем записывать значения величины i после каждого шага алгоритма. Как видно

⁷ "Для того чтобы поверить в алгоритм нужно его увидеть." Дональд Кнут.

⁸ Название «алгоритм» произошло от латинской формы имени величайшего среднеазиатского математика Мухаммеда ибн Муса ал-Хорезми (Alhorithmi), жившего в 783-850 гг. В своей книге «Об индийском счете» он изложил правила записи натуральных чисел с помощью арабских цифр и правила действий над ними «столбиком».

⁹ Простым называется число, не имеющее делителей, отличных от единицы и его самого, причем единица в множество простых чисел не входит.

из приведенного исследования после завершения работы алгоритма переменная i действительно будет содержать наименьший простой делитель исходного числа N .

$N = 3$	$N = 4$	$N = 2$
Шаг 1: $i = 2$	Шаг 1: $i = 2$	Шаг 1: $i = 2$
Шаг 2: $i = 2$	Шаг 2: $i = 2$	Шаг 2: $i = 2$
Шаг 3: $i = 3$		
Шаг 2: $i = 3$		

Основные свойства любого алгоритма – это конечность, определенность, эффективность и массовость. Рассмотрим их последовательно более подробно.

Конечность. Алгоритм должен всегда заканчиваться после выполнения конечного числа шагов. Рассмотренный алгоритм удовлетворяет этому условию, так как величина i вначале меньше N , ее значение увеличивается на единицу к каждому очередному выполнению шага 2, и поэтому выполнение алгоритма будет прекращено на шаге 2 при $i = N$, если N – простое число, или ранее при составном N .

Определенность. Действия, которые необходимо произвести на каждом шаге, должны быть определены строго и однозначно в каждом возможном случае. В данном примере применена достаточно определенная, хотя и не вполне формальная система обозначений. Чаще алгоритмы записывают с использованием более формальных языков программирования, в которых каждое утверждение имеет точный смысл.

Эффективность. От алгоритма требуется, чтобы он был эффективным. Это означает, что все операции, которые необходимо произвести в алгоритме, должны быть достаточно простыми, чтобы их в принципе можно было выполнить точно и за конечное время. В рассмотренном алгоритме выполняются лишь следующие операции: сравниваются два целых числа, одно положительное число делится на другое, переменной присваивается значение целого числа два, ее значение увеличивается на единицу.

Понятие эффективности алгоритма имеет и свои количественные характеристики. Различают временную и емкостную эффективности. Первая из них характеризует время выполнения алгоритма, а вторая – требуемый для этого объем памяти.

Массовость. Алгоритм составляется в общем виде, т.е. он должен быть применим для некоторого класса задач, различающихся лишь исходными данными. При этом исходные данные могут выбираться из некоторой области, которая называется *областью применимости алгоритма*. В случае рассмотренного ранее алгоритма, областью применимости является множество натуральных чисел, больших единицы.

В зависимости от поставленных целей алгоритм можно представить либо в текстовой форме, либо в графической форме, либо в алгоритмической форме.

Очевидно, что словесный способ описания алгоритмов (в текстовой форме) применим только для простейших задач. Описание алгоритмов на

языке программирования (в алгоритмической форме) будет рассмотрено в следующих разделах. Остановимся более подробно на графическом представлении алгоритма.

Одним из самых наглядных форм представления алгоритма является графическое представление, которое принято называть *блок-схемой*. Отдельные действия алгоритма в этом случае изображаются при помощи различных геометрических фигур (блоков), а связи между ними указываются стрелками (рис.1.13). Блоки сопровождаются надписями. Конфигурация и размеры блоков, а также порядок построения схем определяется стандартом, которого следует придерживаться при окончательном оформлении программной документации.



Рис.1.13 – Основные элементы блок-схем

1.7.2 Основные алгоритмические структуры

Существуют три элементарные алгоритмические структуры: линейная, ветвления и повторения. Все другие алгоритмические структуры образуются от элементарных структур.

Алгоритмическая линейная структура – это структура алгоритма, в котором все действия выполняются последовательно одно за другим (рис.1.14).

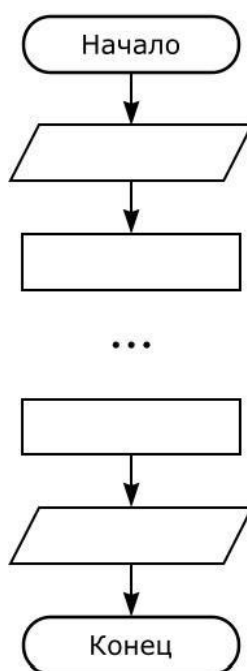


Рис. 1.14 – Размещение блоков в линейной алгоритмической структуре

Рассмотрим несколько примеров линейных алгоритмов.

Пример 1.1. По известным длинам трех сторон треугольника вычислите площадь и периметр треугольника.

Пусть a, b, c – длины сторон треугольника. Требуется найти P – периметр и S – площадь треугольника. Для нахождения площади можно воспользоваться формулой Герона: $S = \sqrt{r(r-a)(r-b)(r-c)}$, где r – полупериметр.

Входные данные: a, b, c .

Выходные данные: S, P .

Блок-схема алгоритма представлена на рис.1.15.

Пример 1.2. Известны плотность и геометрические размеры цилиндрического слитка, полученного в металлургической лаборатории. Найти объем, массу и площадь основания слитка.

Входные данные: R – радиус основания цилиндра, h – высота цилиндра, ρ – плотность материала.

Выходные данные: m – масса слитка, V – объем слитка, S – площадь основания.

Блок-схема алгоритма представлена на рис.1.16.

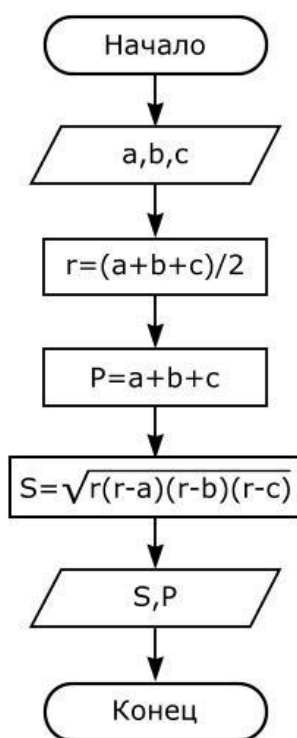


Рис. 1.15 – Блок-схема алгоритма примера 1.1

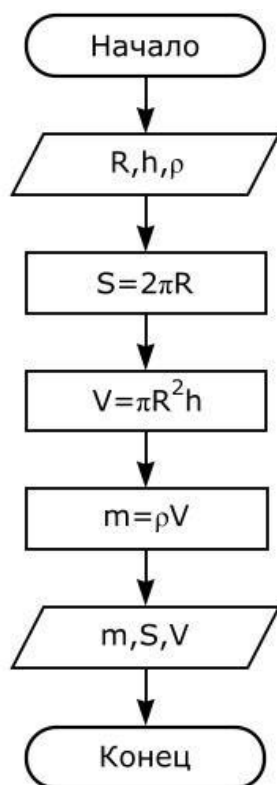


Рис. 1.16 – Блок-схема алгоритма примера 1.2

Алгоритмической структурой ветвления – называется структура алгоритма, которая позволяет исполнителю выбрать сценарий дальнейших действий в зависимости от истинности определенного условия. В отличие от линейных алгоритмов, в которых команды выполняются последовательно одна за другой, в алгоритмические структуры ветвления входит условие, в зависимости от истинности условия выполняется та или иная последовательность команд

Рассмотрим несколько примеров построения алгоритмов разветвленной структуры.

Пример 1.3. Известны коэффициенты квадратного уравнения. Вычислить действительные корни квадратного уравнения.

Пусть a, b, c – коэффициенты квадратного уравнения. Дискриминант квадратного уравнения вычисляется по формуле: $d = b^2 - 4ac$. Действительные корни квадратного уравнения вычисляются по следующим формулам: если $d > 0$, то $x_1 = \frac{-b - \sqrt{d}}{2a}$, $x_2 = \frac{-b + \sqrt{d}}{2a}$; если $d = 0$, то $x = -b/2a$.

Входные данные: a, b, c .

Выходные данные: x, x_1, x_2 .

Блок-схема алгоритма представлена на рис.1.17.

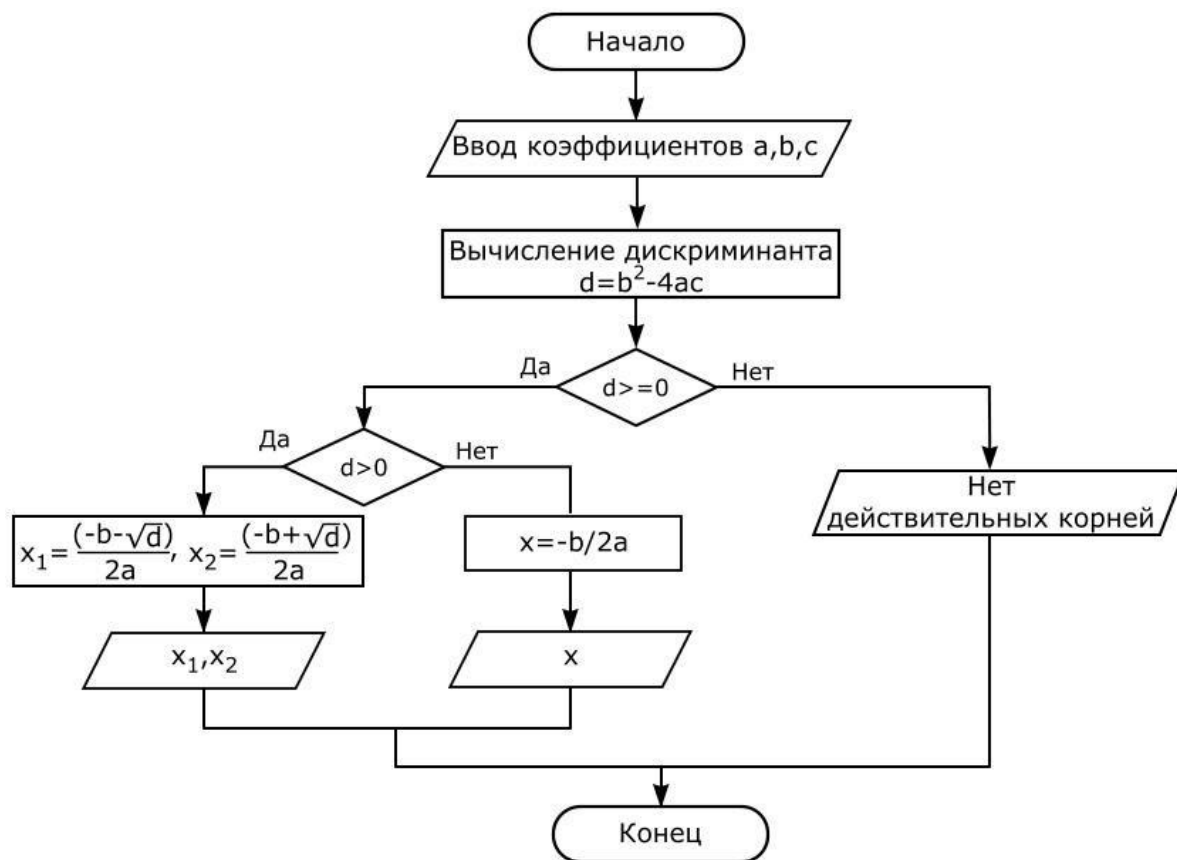


Рис. 1.17 – Блок-схема алгоритма примера 1.3

Алгоритмическая структура повторения – это структура алгоритма, которая дает исполнителю указание повторять некоторые действия, пока определенное условное утверждение истинно. Структуру повторения часто называют *циклической структурой*.

Пример 1.4. Построим блок-схему для рассмотренной в п.1.7.1 задачи нахождения наименьшего простого делителя натурального числа N , большего единицы.

В алгоритме задачи используется структура повторения, которая реализуется шагами 2 и 3. Выполнение указанных в этих шагах действий повторяется до тех пор, пока N не делится нацело на i . Истинность данного утверждения проверяется на шаге 2 каждый раз перед выполнением шага 3.

Входные данные: N .

Выходные данные: i .

Блок-схема алгоритма представлена на рис.1.18.

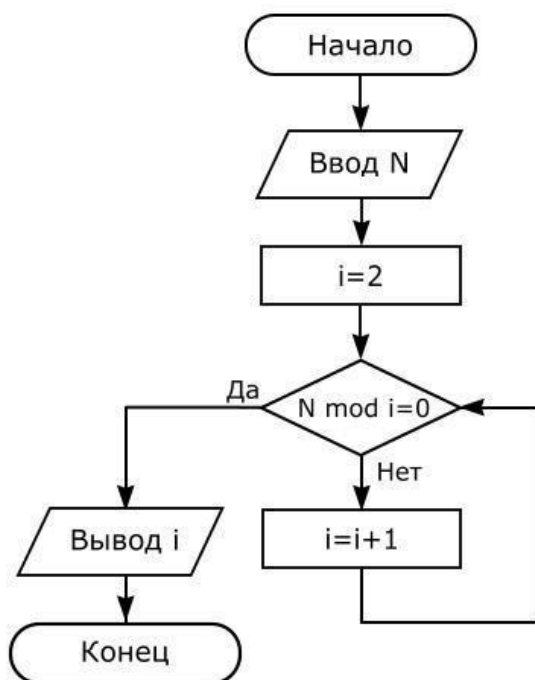


Рис.1.18 – Блок-схема алгоритма примера 1.4

Упражнения к главе 1

1. Программист написал программу на языке машинных команд, которая содержит 21 команду. Каждая команда занимает 4 байта памяти. Программа размещается в памяти, начиная с нулевого адреса. Указать шестнадцатеричный адрес последней команды в программе.

Ответ: 53.

2. Сколько символов содержит сообщение, записанное с помощью 16-ти символьного алфавита, если его объем составил 1/16 часть Мбайта?

Ответ: 32768 символов.

3. Сообщение, закодированное с помощью четырёхсимвольного кода, передавалось 30 секунд со скоростью 100 симв/мин. Каков объем этого сообщения в байтах?
Ответ: 25 байт.
4. Какое наибольшее десятичное число можно записать тремя цифрами в системе счисления: а) двоичной; б) восьмеричной; в) шестнадцатеричной?
Ответ: 7, 511, 4095.
5. В какой системе счисления справедливо равенство $21 + 24 = 100$?
Решение. Пусть x – искомое основание системы счисления. Тогда $100 = 1 \cdot x^2 + 0 \cdot x^1 + 0 \cdot x^0 = x^2$, $21 = 2 \cdot x^1 + 1 \cdot x^0$, $24 = 2 \cdot x^1 + 4 \cdot x^0$.
Таким образом, $x^2 = 4x + 5$, или $x^2 - 4x - 5 = 0$. Положительным корнем этого квадратного уравнения является $x=5$.
Ответ: числа записаны в пятеричной системе счисления.
6. Десятичное число 59 эквивалентно числу 214 в некоторой другой системе счисления. Найдите основание этой системы.
Ответ: число записано в пятеричной системе счисления.
7. Переведите числа из десятичной системы в двоичную, восьмеричную, шестнадцатеричную, а затем проверьте результаты, выполнив обратные переводы:
а) 12510; б) 22910; в) 8810; г) 37,2510; д) 206,12510.
8. Переведите в двоичную и восьмеричную системы шестнадцатеричные числа:
а) 2CE16; б) 9P4016 в) ABCDE16; г) 1010,10116; д) 1ABC,9116.
9. Запишите числа в прямом коде (формат 1 байт):
а) 31; б) -63; в) 65; г) -128. Для всех ли чисел это возможно?
10. Запишите числа в обратном и дополнительном кодах (формат 1 байт):
а) -9; б) -15; в) -127; г) -128.
11. Найдите десятичные представления чисел, записанных в дополнительном коде:
а) 1 1111000; б) 1 0011011; в) 1 1101001; г) 1 0000000.
12. Найдите десятичные представления чисел, записанных в обратном коде:
а) 1 1101000; б) 1 0011111; в) 1 0101011; г) 1 0000000.
13. Вычислите минимальные и максимальные по модулю конечные вещественные числа, которые изображаются с помощью:
а) 4 байт с $d=8$, $r=23$; б) 8 байт с $d=11$, $r=52$; в) 10 байт с $d=16$, $r=63$.
14. Запишите алгоритм обмена значениями между двумя однотипными числовыми переменными при условии, что:
а) можно использовать третью переменную;
б) третью переменную использовать нельзя.
15. Построить блок-схему алгоритма, определяющего есть ли среди введенных трех целых чисел одинаковые числа.

16. Построить блок-схему алгоритма, определяющего максимальное число среди введенных трех целых чисел.
17. Построить блок-схему вычисления факториала натурального числа.
18. Построить блок-схему алгоритма, решающую следующую задачу: дано количество секунд с начала суток. Вычислить текущее время в виде количества часов, минут и секунд.
19. Построить блок-схему алгоритма, решающую следующую задачу: даны три вещественных числа a , b , c . Существует ли треугольник со сторонами a , b , c ?
20. Построить блок-схему алгоритма, вычисляющего сумму квадратов всех натуральных чисел от 1 до N .

2 ЭЛЕМЕНТЫ ЯЗЫКА JAVA

2.1 Краткая история создания Java

Язык Java был создан программистами компании Sun Microsystems, Inc. в 1992 году. Первоначально он назывался «Оак» (Дуб), но в 1995 году был переименован в Java. Стимулом к созданию Java послужила назревшая к тому времени потребность в архитектурно-независимом языке. Существовавшие на тот момент языки программирования были ориентированы на конкретный процессор или требовали наличия соответствующего компилятора. Новый язык должен был решить эту проблему. Предполагалось, что в дальнейшем Java будет использоваться для программирования процессоров, встроенных в различные бытовые устройства.

В результате был создан универсальный объектно-ориентированный язык программирования, позволяющий создавать программы, не зависящие от процессора и операционной системы, с поддержкой современных технологий. Многие важные свойства, реализованные в Java, нашли применение в программировании для, набирающей в то время обороты, сети Интернет. Да и сам язык оказал большое влияние на развитие Интернет.

Как уже отмечалось в п. 1.6 Java – это язык, компилируемый в байт-код, т.е. Java-программа может быть исполнена на любом процессоре без перекомпиляции, нужен только соответствующий интерпретатор, понимающий байт-коды. Этим достигается переносимость языка между разными платформами, которая отражена и в девизе Java: “Write once, run anywhere”¹⁰.

Тот факт, что программа Java интерпретируется, также помогает сделать ее безопасной. Поскольку выполнение каждой программы Java находится под управлением виртуальной машины Java (JVM), то JVM может, как содержать программу, так и оберегать ее от генерации побочных эффектов вне системы. Например, в JVM можно встроить средства проверки, на вредоносность интерпретируемого кода или наличие в нем вирусов. В том случае, если будет обнаружена угроза для системы, JVM не будет выполнять такой код.

Сегодня Java – это не только один из самых популярных компьютерных языков в мире, используемый для разработки веб-ориентированных приложений, но и базовая технология, на которой основываются многие современные программы (в том числе утилиты, игры и бизнес-приложения). Кроме того, благодаря Java стала возможной

¹⁰ с англ. "Написано однажды – работает везде".

современная революция смартфонов, поскольку язык используется для написания программ для платформы Android.

В технологии Java существует несколько основных направлений:

- Java Enterprise Edition (сокращенно Java EE или J2EE) – технология для создания программного обеспечения уровня больших корпораций.

- Java Standard Edition (сокращенно Java SE или J2SE) – технология создания пользовательских приложений.

- Java Micro Edition (сокращенно Java ME или J2ME) – технология создания приложений для мобильных телефонов, карманных персональных компьютеров и других маломощных вычислительных систем.

- Java Card – технология для устройств типа смарт-карт.

В учебнике будет использована технология Java Standard Edition, то есть стандартный дистрибутив Java.

Java постоянно изменяется и адаптируется к изменениям в среде программирования. На момент написания учебника новейшим выпуском Java является Java SE 7, с комплектом разработчика Java JDK 7 и внутренним номером версии 1.7. Это последний главный выпуск Java от Sun Microsystems. В 2010 году компания Sun Microsystems была приобретена корпорацией Oracle.

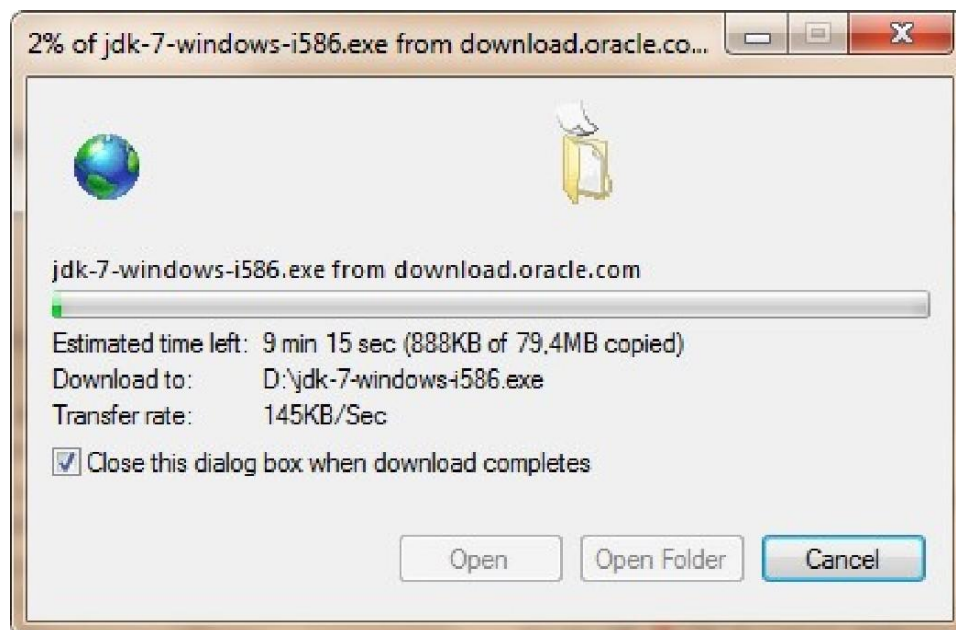
2.2 Установка среды Java

Для того чтобы начать изучение Java и выполнять примеры, которые рассматриваются в дальнейшем, необходимо наличие установленного специального программного обеспечения – Java Development Kit (JDK). Это программное обеспечение можно бесплатно загрузить с веб-сайта компании Oracle¹¹. Рассмотрим подробно процесс установки JDK.

Перейдите на сайт компании Oracle. Выберите в таблице выпуск Java SE 7 или более новый. Нажмите кнопку Download под заголовком JDK в строке таблицы с этим выпуском. Примите условия лицензионного соглашения. Выберите пакет, соответствующий операционной системе, которая установлена на вашем компьютере. Например, для Windows это будет Windows x86 (32- разрядная версия) или Windows x64 (64- разрядная версия).

Щелкните по ссылке с выбранным пакетом. Начнется загрузка файла. На компьютерах с операционной системой Windows это может выглядеть так:

¹¹ <http://www.oracle.com/technetwork/java/javase/downloads/index.html>



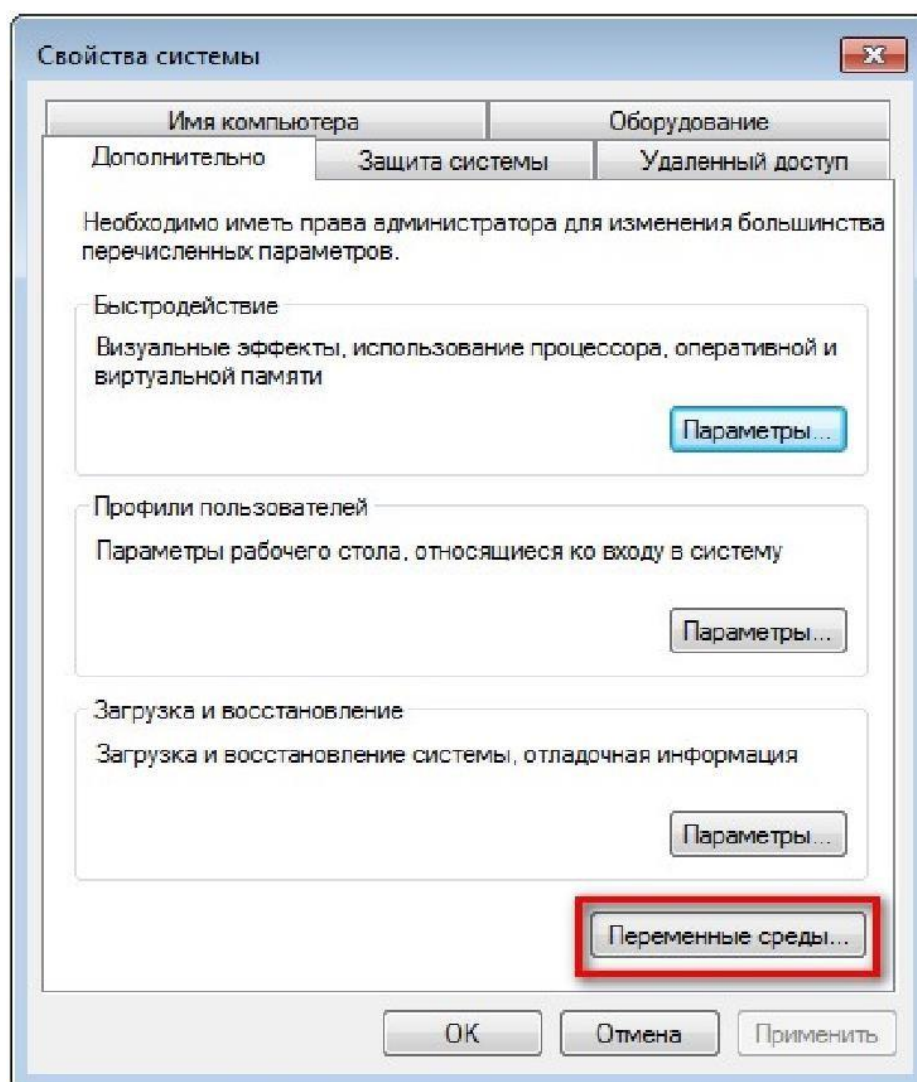
После окончания загрузки запустите процесс установки – дважды щелкните кнопкой мыши по загруженному файлу. Произойдет запуск мастера установки. Например, на компьютере под управлением Windows будет создан следующий каталог: C:\Program Files\Java\jdk1.7.0, где C: – это имя жесткого диска. Можете выбрать другой диск.



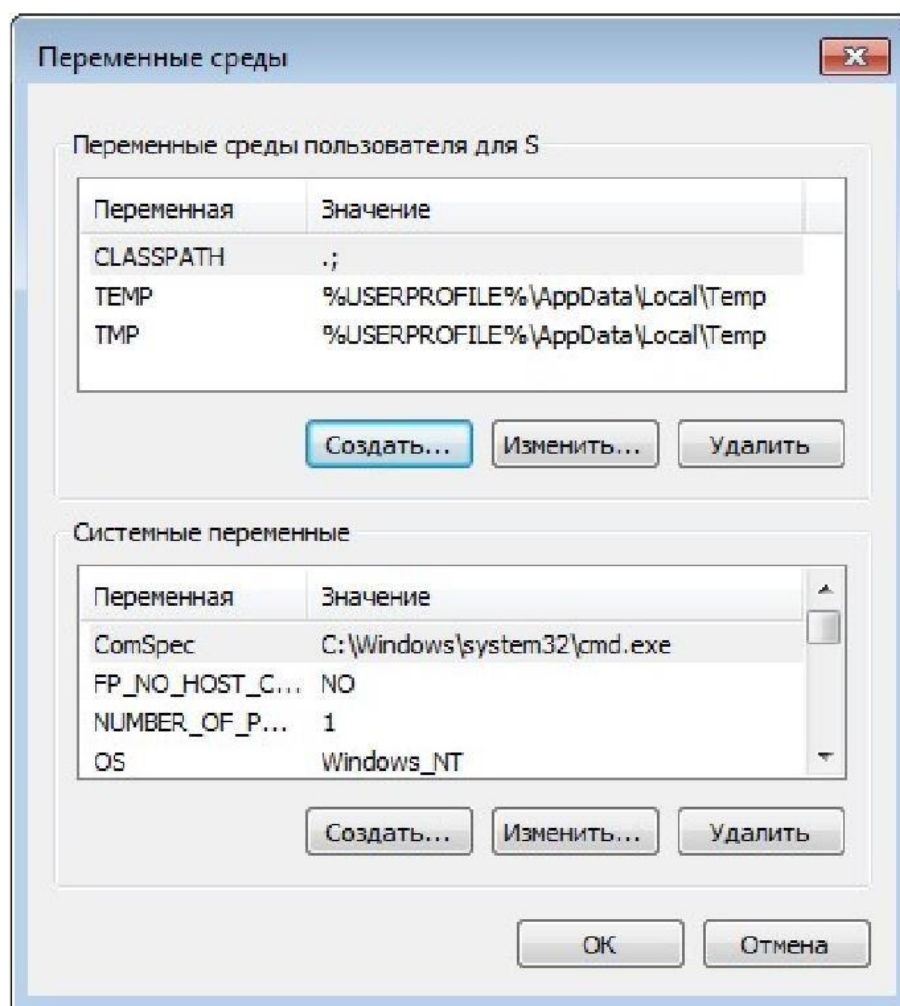
Следуйте указаниям мастера установки – просто нажимайте кнопки *Next*, *Install* и *Finish* в появляющихся на экране окнах. В течение нескольких минут установка Java на компьютере будет завершена.

На следующем этапе установки необходимо задать две *системные переменные*. Например, в Windows нажмите кнопку *Пуск*, далее щелкните правой кнопкой мыши по пункту *Мой компьютер* и выберите пункт *Свойства*.

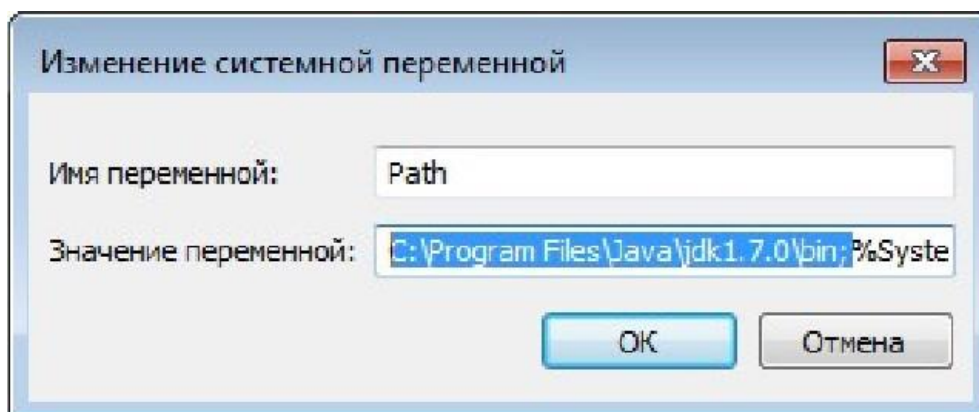
Далее перейдите по ссылке *Дополнительные параметры системы*, которая расположена в левой верхней части панели слева. Перейдите на вкладку *Дополнительно* и нажмите на кнопку *Переменные среды...*



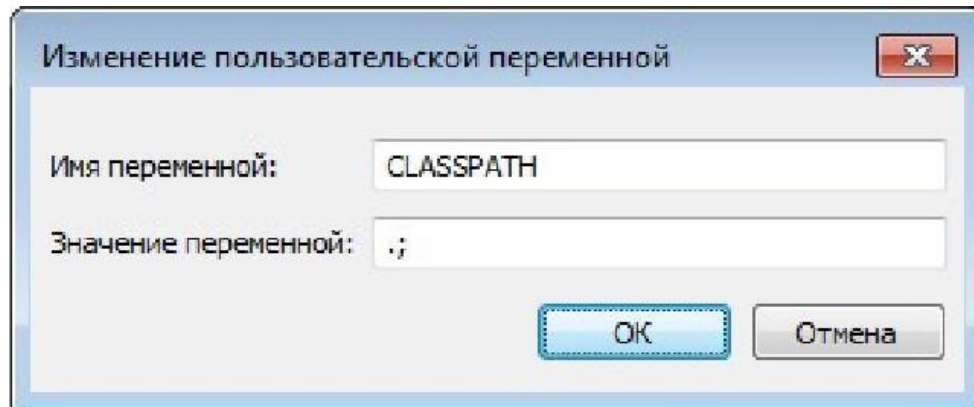
Для Windows 7 окно, которое содержит все системные переменные, имеющиеся в системе, выглядит следующим образом:



Нажмите нижнюю кнопку *Создать* и объявите переменную Path, которая поможет Windows (или Unix) найти пакет JDK на вашем компьютере. Тщательно проверьте путь к каталогу, в который была произведена установка Java. Если переменная Path уже существует, просто добавьте путь к каталогу Java и точку с запятой в самое начало поля *Значение переменной*:



Также объявите переменную `CLASSPATH`. В качестве ее значения укажите точку, за которой следует точка с запятой (`;`). Данная системная переменная поможет Java найти ваши программы. Точка означает, что Java начнет поиск ваших программ в текущем каталоге. Точка с запятой всего лишь играет роль разделителя, за которыми могут идти другие каталоги.



На этом установка Java окончена.

2.3 Выполнение Java-программы

Рассмотрим подробнее этапы выполнения Java-программы (рис.2.1). Программы, написанные на языке Java, представляют собой исходный код, который хранится в файлах с расширением *java*. Компилятор Java преобразует этот исходный код в байт-код, который хранится в файлах с расширением *class*. Байт-код в свою очередь интерпретируется виртуальной машиной Java. А уже виртуальные машины Java, реализованные отдельно для каждой платформы (например, Linux, Solaris, Windows, MacOS) обеспечивают кросс-платформенность или независимость кода от платформы, на которой он будет исполнен.

Таким образом, программу Java, которая должна работать на нескольких платформах, нет необходимости компилировать для каждой платформы отдельно. Можно откомпилировать и отладить программу на одной из платформ и в результате получить байт-код, пригодный для любой платформы, где есть виртуальная машина Java.

Компилятор Java реализован в виде утилиты *javac*, а виртуальная машина Java – в виде программы *java*.

Итак, чтобы создать работающую программу сначала нужно создать исходный файл с текстом программы на языке Java и сохранить его на диск. Для написания программы можно использовать любой текстовый редактор, например, «Блокнот». Главное, чтобы итоговый текстовый файл был сохранен с расширением *java*.

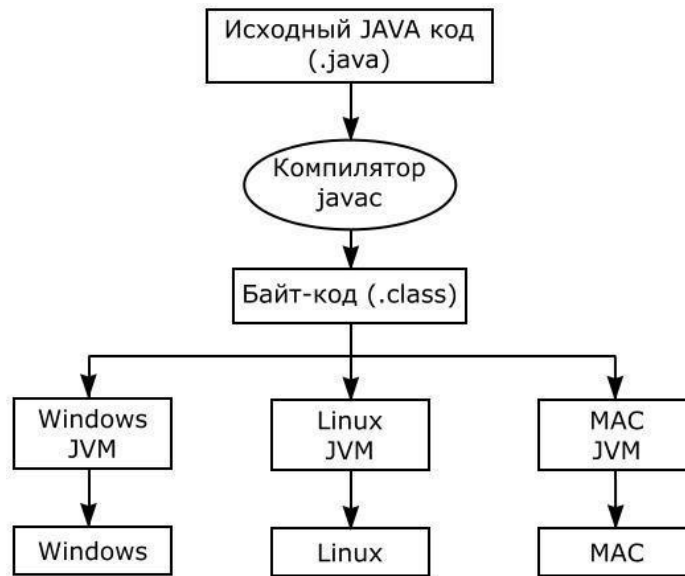


Рис.2.1 – Схема интерпретации байт-кода Java

В качестве примера наберем в редакторе «Блокнот» текст программы на языке Java, которая выводит на экран слова Hello World¹²:

```
class HelloWorld {  
    public static void main (String[] args) {  
        System.out.println("Hello World"); }  
}
```

Сохраните этот файл с именем HelloWorld.java¹³.

Откомпилируем файл, используя, входящий в состав пакета JDK, компилятор `javac`. Перейдем в командную строку DOS. Для этого нажмите кнопку *Пуск*, введите в строке поиска команду `cmd` и нажмите кнопку *ОК*. Измените текущий каталог на тот, в котором сохранен исходный файл. Например, если файл сохранен в каталоге `C:\MyFiles`, то следует выполнить команду `cd c:\MyFiles` и нажать клавишу *Enter*. Откомпилируем программу:

```
C:\MyFiles>javac HelloWorld.java
```

Если компиляция прошла успешно, то вы не увидите каких-либо сообщений. Если же в программе есть синтаксические ошибки, то компилятор выведет сообщение об ошибке. В этом случае необходимо

¹² Пока просто наберем текст программы, более подробно назначение каждой строки исходного кода будет рассмотрено в конце данного параграфа.

¹³ Будьте внимательны, файл должен быть назван также как и `class` в первой строке программы. В имени не должно быть пробелов. Язык Java чувствителен к регистру букв.

будет исправить все ошибки в исходном файле и откомпилировать программу заново.

После успешной компиляции в каталоге с файлом `.java` появится новый файл с именем `HelloWorld.class`.

Для того чтобы запустить программу, введем в командной строке следующую команду.

```
C:\MyFiles>java HelloWorld  
Hello World
```

Используемая в командной строке программа `java` входит в среду выполнения *JRE* (Java Run-time Environment) и запускает JVM, в которую и загружает программу `HelloWorld`.

В результате успешного выполнения программы, на экране в командном окне появится выводимый текст: `Hello World`.

Теперь можете изменить программу так, чтобы она выводила другой текст, например, вашу фамилию и имя.

Как работает программа HelloWorld

Рассмотрим структуру нашей первой программы (рис.2.2). Любая программа на Java представляет собой *класс* (англ. *class*). Программа может состоять и из нескольких классов, но в самом простом случае это должен быть, по крайней мере, один класс. В этом классе записывается текст программы. Начало класса отмечается служебным словом `class`, за которым следует имя класса, выбираемое произвольно, в данном случае `HelloWorld`. Все, что содержится в классе, записывается в фигурных скобках (блоке) и составляет *тело класса* (англ. *class body*).

Все действия, которые должна выполнять программа, производятся с помощью методов обработки информации. *Методы* (англ. *method*) различаются по именам. Один из методов обязательно должен называться `main()`, он является точкой входа *приложения* на Java. Наличие метода `main()` в ваших приложениях является обязательным условием, так как с него начинается выполнение программы. То, что `main` – это метод, говорят круглые скобки после слова `main`. Все, что содержит метод – тело метода (англ. *method body*), записывается в фигурных скобках. Каждый метод начинается со *строки объявления* (или заголовка), которую называют *сигнатурой метода*. Как вы можете видеть, у главного метода сложная запись строки объявления: `public static void main (String[] args)`. Позже мы разберем назначение каждого слова, а пока следует запомнить, что сигнатура главного метода записывается именно так и никак иначе.

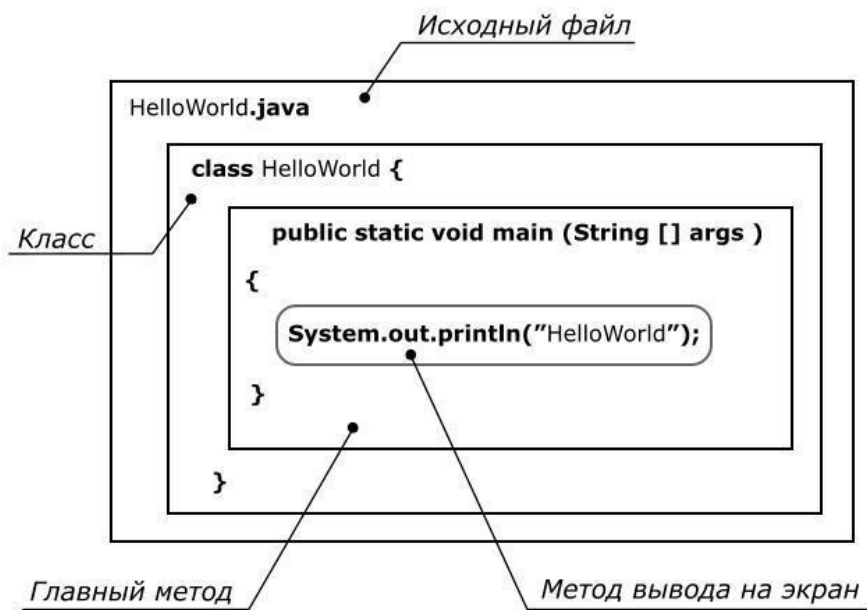


Рис.2.2 – Структура программы HelloWorld

Методы могут *вызывать* (использовать) другие методы, например, в нашей программе метод `main()` чтобы напечатать на экране текст `Hello World`, вызывает метод `println()`. Метод `println()` выполняет вывод данных (*аргументов*, записанных в его круглых скобках) в системную *консоль* (командную консоль). У метода `println()` один аргумент – текстовая константа `"Hello World"`. Текстовые константы записываются в кавычках, которые являются только ограничителями и не входят в состав текста.

После вывода текста курсор переходит на начало следующей строки экрана, на что указывает окончание `ln`, слова `println` (сокращение слов `print line`). В Java есть еще и метод `print()`, который оставляет курсор в конце выведенной строки.

Каждая команда или вызов метода должны заканчиваться точкой с запятой (;).

2.4 Некоторые интегрированные среды разработки

Как не трудно заметить, писать программу в текстовом редакторе, а потом компилировать и запускать в командной строке не совсем удобно. Обычно программисты используют для этого различные интегрированные среды разработки IDE, которые позволяют в удобном режиме набрать программный код, проверить его (на уровне синтаксических конструкций), откомпилировать и запустить программу на выполнение.

Существует ряд бесплатных IDE средств для программирования на Java. Они отличаются друг от друга интерфейсом, набором возможностей,

требованиями к ресурсам и сложностью в освоении. Ниже перечислены несколько популярных IDE, которые вы можете использовать для создания программ, и ссылки на сайты, с которых их можно свободно загрузить. Вполне возможно, что на разных этапах изучения Java, по мере приобретений знаний и опыта программирования, вы будете использовать разные среды разработки.



BlueJ – интерактивная среда разработки программ на языке Java, созданная специально для обучения программированию на начальном уровне, но также подходит для разработки небольших программ. BlueJ разрабатывается и поддерживается совместной

исследовательской группой университетов La Trobe University (Мельбурн, Австралия) и University of Kent (Кентербери, Великобритания) при поддержке компании Oracle. Дизайн BlueJ значительно отличается от других сред разработки. Среда имеет ясный, простой интерфейс пользователя и позволяет легко адаптироваться и быстро начать программировать начинающим программистам. Скачать BlueJ можно по ссылке <http://www.bluej.org/download/download.html>. Инструкция по применению BlueJ на русском языке доступна по ссылке <http://www.bluej.org/tutorial/tutorial-russian.pdf>.



Eclipse IDE – интегрированная среда разработки с открытым исходным кодом, один из лучших инструментов Java, созданных за последние годы. Eclipse разработана для широкого круга операционных систем, таких как Linux, Microsoft Windows и Mac OS. Поддержкой и разработкой Eclipse в настоящее время занимается организация Eclipse

Foundation и сообщество Eclipse. Eclipse IDE for Java Developers можно загрузить с официального сайта в сети Интернет <http://www.eclipse.org/downloads/>.

Eclipse является модульной средой разработки на основе программного ядра и интерфейсов для написания и подключения дополнительных модулей (плагинов, plugin). Таким образом, среда Eclipse может быть использована не только для написания приложений на языке Java, но также и для других целей, в зависимости от установленных плагинов. При первоначальном знакомстве со средой IDE Eclipse она может показаться несколько сложной для неподготовленного пользователя. Подробнее о работе в Eclipse можно узнать из Eclipse IDE Tutorial по ссылке <http://www.vogella.com/articles/Eclipse/article.html>.



NetBeans IDE – многофункциональная открытая интегрированная среда разработки для создания мобильных, настольных и Web-приложений на базе Java и ряда других языков. Проект NetBeans IDE поддерживается и спонсируется компанией Oracle,

однако разработка NetBeans ведется независимым сообществом разработчиков-энтузиастов (NetBeans Community) и компанией NetBeans Org.

По качеству и возможностям последние версии NetBeans IDE не уступают лучшим коммерческим (платным) интегрированным средам разработки для языка Java. NetBeans поддерживает плагины, позволяя разработчикам расширять возможности среды. В настоящее время NetBeans доступна для различных операционных платформ и предлагается в версиях на разных языках, включая русский. Загрузить NetBeans IDE можно по ссылке <http://netbeans.org/downloads/>. Инструкция по установке доступна по ссылке http://netbeans.org/community/releases/72/install_ru.html.



IntelliJIDEA IDE – интегрированная среда разработки программного обеспечения на многих языках программирования, в частности Java, JavaScript, Python и др., разработанная компанией JetBrains. Доступна в двух версиях: Community Edition и Ultimate Edition. Community Edition является полностью свободной версией, доступной под лицензией Apache 2.0. В ней реализована полная поддержка Java SE, Groovy, Scala, а также интеграция с наиболее популярными системами управления версиями. В версии Ultimate Edition реализована поддержка Java EE, UML-диаграмм, подсчет покрытия кода, а также поддержка других систем управления версиями, языков и фреймворков. Загрузить IntelliJIDEA IDE для операционной системы Windows можно по ссылке <http://www.jetbrains.com/idea/download/index.html>.

2.5 Лексические основы

2.5.1 Лексемы, комментарии и пробелы

Простейшие слова или минимальные значащие единицы, из которых состоит текст Java-программы, называются *лексемами* (англ. tokens). В общем случае любая программа представляет собой набор пробелов, комментариев и лексем.

Пробелы используются для разделения лексем. В данном случае пробелом будем называть как сам символ пробела, так и знаки табуляции и перевода строки.

Java – язык, который допускает произвольное форматирование текста программ. Программист при желании может записать всю программу одной строкой. Однако форматирование с помощью пробелов существенно облегчает чтение и понимание программного кода.

Комментарий – это поясняющий текст в программе, который не учитывается компилятором и не выполняется. Существует три

разновидности комментариев: комментарии в одной строке (строчные), комментарии в нескольких строках (блочные) и, наконец, комментарии для документирования.

Строчные комментарии начинаются с символов `//` и длятся до конца текущей строки. Как правило, они используются для пояснения именно к данной строке кода, например:

```
counter = 0; // это переменная – счетчик
```

Блочные комментарии могут занимать произвольное количество строк и располагаются между символами `/*` и `*/`, например:

```
/*  
Выводим на экран значение счетчика  
и переводим курсор на новую строку  
*/  
System.out.println(counter);
```

Комментарии для документирования применяется для автоматического создания документации кода в HTML формате с помощью утилиты `javadoc`. Комментарии для документирования начинаются с символов `/**` и заканчиваются символами `*/`. Подробное описание утилиты `javadoc` можно найти в технической документации языка на сайте <http://docs.oracle.com/javase/>.

2.5.2 Виды лексем

Как уже отмечалось выше, программа на языке Java – это набор пробелов, комментариев и лексем. Причем при выполнении компиляции программного кода все пробелы и комментарии отбрасываются, они служат только для форматирования и представления исходного текста в удобном для программиста виде. Остаются одни лексемы, которые разделяются на следующие виды: разделители, ключевые слова, идентификаторы, литералы, операторы.

Разделители (англ. `separators`)– это специальные символы, которые используются для группировки и упорядочения текста Java-программы. К разделителям относятся следующие конструкции:

() { } [] ; , .

Некоторые разделители вы уже встречали в тексте программы `HelloWorld`, другие – будут рассмотрены нами в дальнейшем при изучении синтаксиса языка Java.

Ключевыми словами (англ. `key words`) называются последовательности символов, которые зарезервированы для специальных

целей. Ключевые слова могут применяться только по назначению, и их нельзя использовать в качестве имен переменных, классов или методов. Перечень ключевых слов языка Java представлен в табл.2.1.

Идентификаторы (англ. identifiers) – это имена, которые даются различным элементам языка (например, классам, методам, переменным) для упрощения доступа к ним. Например, идентификатором является имя класса – *HelloWorld* и имя метода – *main*.

Идентификаторы могут иметь любую длину и состоять из строчных и прописных букв, цифр, символов подчеркивания `_` и доллара `$` (последний применять не рекомендуется, так как Java использует его в определенных случаях для своих нужд). Идентификатор не может начинаться с цифры. В качестве идентификаторов не могут использоваться ключевые слова Java (а также литералы `true`, `false` и `null`).

Таблица 2.1 – Перечень ключевых слов языка Java

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	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

Обратите внимание, что Java различает строчные и прописные буквы, идентификаторы `main`, `Main`, `MAIN` являются для компилятора Java разными именами. Свои идентификаторы, которые вы будете давать переменным, классам и методам, можно записывать как угодно, однако между Java-программистами заключено соглашение об именовании. Вот несколько пунктов этого соглашения:

- имена классов начинаются с прописной буквы (если имя содержит несколько слов, то каждое слово начинается с прописной буквы);
- имена методов и переменных начинаются со строчной буквы (если имя содержит несколько слов, то каждое следующее слово начинается со строчной буквы);

– имена констант записываются полностью прописными буквами (если имя состоит из нескольких слов, то между ними ставится знак подчеркивания).

Конечно, эти правила необязательны, но следуя им можно значительно облегчить понимание кода и придать программе характерный для Java стиль.

Литерал (англ. literal) – константа некоторого типа, не имеющая имени, а указанная собственно записью. Литералы позволяют задать в программе значения для числовых, логических, символьных и строковых выражений.

Числовые литералы подразделяются на целые литералы и литералы с плавающей точкой.

Целые литералы – это любые положительные, нулевые и отрицательные целочисленные значения, записанные в десятичной, шестнадцатеричной, восьмеричной или двоичной системах счисления. Восьмеричное число Java распознает по стоящему впереди нулю, шестнадцатеричное – по стоящим впереди символам 0x или 0X, а двоичное – по символам 0b или 0B. Например, число пять можно представить тремя различными способами: 5, 05, 0x5 и 0b101.

В Java допускается использование одного или нескольких символов подчеркивания `_` в качестве разделителя между цифрами. Это облегчает чтение больших целочисленных литералов. При компиляции символы подчеркивания в литерале игнорируются. Подчеркивание не должно стоять в начале или в конце числа, а в шестнадцатеричном и двоичном представлении сразу же после суффиксов 0x или 0b. Примеры целых литералов¹⁴: 45, -124, 00, 077, 0xDada_Cafe, 0x00_FF__00_FF, 0b1111_1001.

Литералы с плавающей точкой – это десятичные значения, у которых есть дробная часть, записанные в обычном или экспоненциальном формате.

В обычном формате разделителем между целой и дробной частями числа является десятичная точка. Причем в записи числа должна быть хотя бы одна цифра в целой или дробной части, (например, 2., .4).

В экспоненциальном формате дополнительно указывается десятичный порядок. Порядок определяется положительным или отрицательным десятичным числом, следующим за символом E или e. Примеры чисел в экспоненциальном формате: 6.022e23, 314159E-05, 2e+100.

Логические литералы могут принимать только два значения – true (истина) и false (ложь). Ключевые слова true и false могут присваиваться

¹⁴ В шестнадцатеричном формате записи целых литералов буквы A, B, C, D, E, F можно записывать в произвольном регистре.

логическим (булевым) переменным или самостоятельно использоваться в логических выражениях.

Символьные литералы – это символы или управляющие последовательности, заключенные в одиночные кавычки или апострофы (например, 'a', '#', '@'). Управляющие последовательности записываются в апострофах с обратной наклонной чертой (табл.2.2).

Код любого символа можно задать в стандарте Unicode (кодировке UTF-16). В этом случае его следует написать в апострофах после обратной наклонной черты и латинской буквы u, четырьмя шестнадцатеричными цифрами, например, '\u0064' – буква d, '\u005A' – буква Z. Символьные литералы представляют собой 16-битовые значения, которые можно преобразовать в целые числа и к которым можно применять различные арифметические операции, например, сложения и вычитания.

Код любого символа с десятичной кодировкой от 0 до 255 можно также задать в кодировке CP1251, записав его не более чем тремя цифрами в восьмеричной системе счисления в апострофах после обратной наклонной черты, например, '\116' – буква N, '\346' – буква Ж.

Таблица 2.2 – Управляющие последовательности

Управляющая последовательность	Описание
'\n'	перевод на новую строку
'\r'	возврат к началу строки
'\f'	перевод на новую страницу
'\b'	возврат на один символ назад
'\t'	горизонтальная табуляция
'\v'	вертикальная табуляция
'\\'	обратная наклонная черта
'\"'	кавычка
'\''	апостроф.
'\u'	начало кодировки символа Unicode

Строковые литералы состоят из нуля или более символов, заключенные в двойные кавычки (" "). Примером строкового литерала является строка "Hello World". Все управляющие последовательности, а также восьмеричные и шестнадцатеричные формы записи символьных литералов, работают точно так же и в строках. Например, можно вывести на экран строковый литерал "Привет" с переводом курсора на новую строку используя символы в кодировке Unicode и управляющий символ перевода строки :

```
System.out.print("\u041F\u0440\u0438\u0432\u0435\u0442\n");
```

Строковые литералы в Java должны начинаться и заканчиваться в одной и той же строке исходного кода. Чтобы записать длинную строку в виде одного строкового литерала, надо после закрывающей кавычки на первой и следующих строках поставить плюс (+), тогда компилятор соберет две (или более) строки в один строковый литерал. В данном случае плюс является не операцией сложения, а так называемой операцией *сцепления*. Например, длинная строка может быть записана таким образом:

```
"This is a "+
    "two-line string"
```

Операторы (англ. operators) – это лексемы, используемые в различных операциях – арифметических, битовых, логических, операциях сравнения и присваивания. В табл.2.3 приведен полный перечень операторов языка Java.

Таблица 2. 3 – Операторы языка Java

=	>	<	!	~	?	:	==	<=	>=
!=	&&		++	--	+	-	*	/	&
	^	%	<<	>>	>>>	+=	-=	*=	/=
&=	=	^=	%=	<<=	>>=	>>>=			

2.6 Примитивные типы данных и переменные

Java – строго типизированный язык. Все типы в Java делятся на примитивные (простые) и ссылочные.

Примитивные типы (англ. primitive types) можно разделить на четыре группы (рис.2.3):

- целые числа – типы byte, short, int, long, которые представляют точные целые числа со знаком;
- числа с плавающей точкой – типы float, double, которые представляют числа, определенные с точностью до определенного десятичного знака;
- символы – тип char, представляет символы символического набора, такие как буквы и цифры.
- булевы значения – тип boolean, предназначенный для представления значений типа “истинно/ложно”.

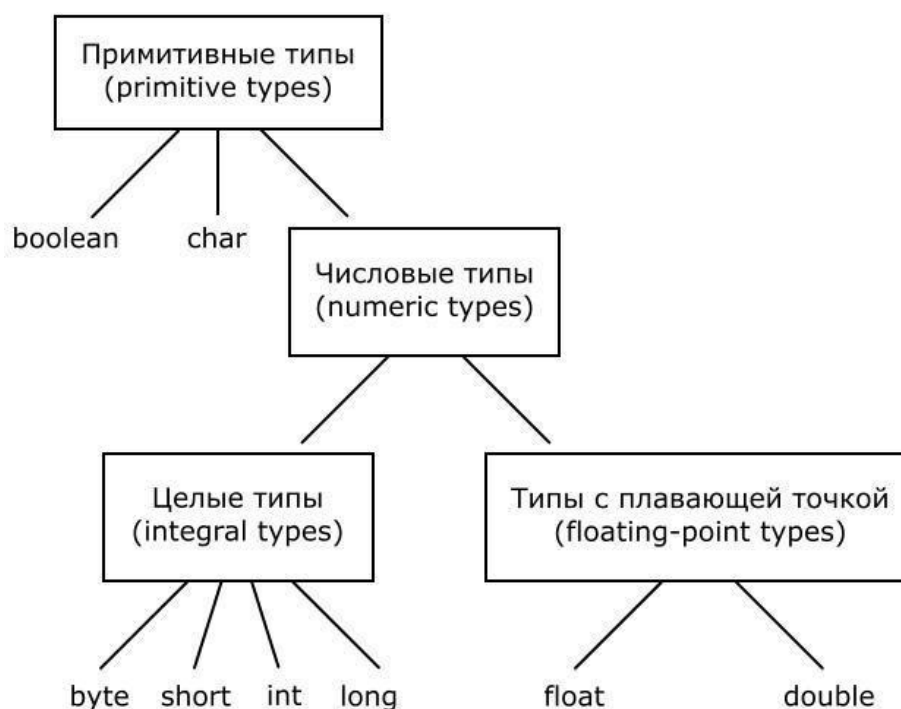


Рис.2.3 – Иерархия примитивных типов языка Java

Все остальные типы Java являются ссылочными типами (англ. reference types). К ним относятся все классы, интерфейсы и массивы.

Все примитивные типы данных обладают строго определенным диапазоном допустимых значений. Для каждого типа определены объемы памяти, необходимые для сохранения соответствующих данных, и множество допустимых операций над этими данными.

2.6.1 Переменные

Переменные (англ. Variables) – именованные участки памяти, в которых программа может хранить данные определенного типа (называемые *значением переменной*) и обращаться к этим данным, используя имя переменной.

Перед тем, как использовать переменную в программе, ее нужно объявить. *Объявление* (англ. declaration) переменной состоит из наименования типа, идентификатора и инициализации. Тип определяет, какие величины может хранить переменная. Идентификатор уникально ее определяет и позволяет обращаться к переменной в программе. Инициализация предполагает задание начального значения переменной.

Основная форма объявления переменных выглядит следующим образом¹⁵:

¹⁵ В квадратных скобках указаны необязательные параметры.

тип идентификатор [=значение] [, идентификатор [=значение]
...];

Переменной можно присвоить начальное значение (инициализировать ее), указывая знак равенства и само значение. Вообще же инициализация является необязательным параметром объявления и в большинстве случаев ее можно отложить. Для объявления нескольких переменных одного типа можно использовать список, разделяемый запятыми. Объявление заканчивается точкой с запятой. Примеры объявлений переменных примитивных типов приведены в табл. 2.4.

2.6.2 Целые типы

К целым типам в Java относятся следующие типы: `byte`, `short`, `int`, `long`. Все эти типы представляют значения со знаком, которые хранятся в двоичном виде с дополнительным кодом, т.е. для отрицательных чисел хранится не их двоичное представление, а дополнительный код этого двоичного представления.

Таблица 2.4 - Примеры объявления переменных

Пример объявления	Описание
<code>boolean f;</code>	Объявление одной переменной без инициализации
<code>double a,b,c;</code>	Объявление списка переменных без инициализации
<code>int a=3, b=5, c;</code>	Объявление списка переменных с инициализацией переменных <code>a</code> и <code>b</code>
<code>int a=b=3;</code>	Переменным <code>a</code> и <code>b</code> присваивается одно и то же значение 3, однако, объявляется лишь первая из них (переменная <code>a</code>), <code>b</code> должна быть уже объявлена ранее.
<code>int a=2*5, b=3;</code> <code>int c=a+b;</code>	Инициализатором переменной является арифметическое выражение. В первом случае выражение будет вычислено во время компиляции, а во втором – при исполнении программы. Результат выполнения выражения должен быть совместим с переменной по типу.

Для каждого целого типа определена длина (количество байтов, выделяемых для хранения значений типа в оперативной памяти) и диапазон значений (табл.2.5). В отличие от диапазона значений, который

строго определен, длина типов условна и на разных компьютерах может отличаться от указанной в таблице.

Таблица 2.5 – Длина и диапазон допустимых значений целых типов

Тип	Длина (в байтах)	Диапазон допустимых значений
byte	1	от -128 до 127
short	2	от -32768 до 32767
int	4	от -2147483648 до 2147483647
long	8	от -9223372036854775808 до 9223372036854775807

Наиболее часто используемым целочисленным типом является `int`. Это обусловлено тем, что при указании значений типа `byte` и `short` в выражениях их тип повышается до типа `int` при вычислении выражения (о повышении типов в выражениях будем говорить позже).

В случае, когда диапазона типа `int` недостаточно для хранения требуемого значения, используют тип `long`. Большой диапазон допустимых значений типа `long` позволяет программисту работать с большими целыми числами.

Примеры объявления переменных целых типов:

```
byte b1 = 50, b2 = 0b101011, b3;  
short sh = -30_254;  
int i = -100, j = 100, k = 10_555;  
long ln = 125, veryBig = 2147483648L;
```

Целочисленные литералы хранятся в формате типа `int`. Однако когда значение литерала присваивается переменной типа `byte` или `short`, ошибки несоответствия типа не будут возникать, если значение литерала находится в диапазоне допустимых значений данного типа.

В том случае, когда необходимо присвоить переменной типа `long` целочисленный литерал, значение которого выходит за границы диапазона типа `int`, в конце литерала можно записать прописную или строчную букву `L`, тогда он будет сохранен в длинном формате типа `long`¹⁶: `25L`, `-057L`, `0xffL`.

¹⁶ Не используйте при записи длинных целых литералов строчную латинскую букву `l`, ее легко спутать с единицей.

2.6.3 Типы с плавающей точкой

В Java существуют два типа с плавающей точкой: `float` и `double`, которые соответственно представляют числа одинарной и двойной точности. Эти типы также называют вещественными типами. Их длины и области допустимых значений регламентированы стандартом IEEE-754-1985 и описаны в табл. 2.6.

Таблица 2.6 – Длина и диапазон допустимых значений типов с плавающей точкой

Тип	Длина (в байтах)	Приблизительный диапазон допустимых значений
<code>float</code>	4	от 1.4e-045 до 3.4e+038
<code>double</code>	8	от 4.9e-324 до 1.8e+308

По умолчанию в Java литералам с плавающей точкой присвоен тип `double`. Чтобы указать литерал типа `float`, к нему нужно дописать символ `F` или `f`. Литерал типа `double` можно также указать явно, дописывая к нему символ `D` или `d`, но это излишне.

Примеры объявления и инициализации вещественных типов:

```
float a = 0.057f, b = -3.14f;  
double c = -678.045, d=.786, e=52.;
```

Стандарт IEEE 754 включает в себя не только положительные и отрицательные вещественные числа, но и положительные и отрицательные нули, положительные и отрицательные бесконечности (`POSITIVE_INFINITY` и `NEGATIVE_INFINITY`), и специальное значение «Не число» (Not a Number, NaN). Значение NaN используется для представления результатов некоторых недопустимых операций, таких как деление нуля на ноль. Существуют константы `Float.NaN` и `Double.NaN` для типа одинарной и двойной точности соответственно.

Числа с плавающей точкой не желательно использовать в вычислениях, где ошибки округления недопустимы, например, в финансовых вычислениях. Так в результате выполнения команды `System.out.println(2.-1.1)` будет выведено не 0,9 как логично ожидать, а 0,8 9999999999999999. Подобные ошибки связаны с внутренним двоичным представлением чисел. Подобно тому, как в десятичной системе счисления нельзя точно представить результат деления 1/3, так и в двоичной системе невозможно точно представить 1/10. Если необходимо исключить ошибки округления, то следует использовать класс `BigDecimal`.

2.6.4 Символьный тип

В Java для хранения символов используется тип данных `char`. Для представления символов в нем используется кодировка Unicode с фиксированным размером в 16 бит. Диапазон допустимых значений – от 0 до 65536 (соответственно 2^{16}). Не существует отрицательных значений типа `char`. Первые 127 значений набора символов Unicode занимает набор символов ASCII.

Отметим, что хотя в формальной спецификации Java тип `char` упоминается как целый и находится в той же категории, что и типы `byte`, `short`, `int`, `long`, все же целесообразно рассматривать его как отдельный символьный тип, поскольку основное назначение типа `char` – представлять символы Unicode.

Примеры объявления символьных переменных:

```
char ch1= 'a', nl='\n';  
char ch2=88; // код переменной X
```

Тип `char` можно также использовать как целочисленный тип при выполнении арифметических операций. Например, он позволяет выполнять увеличение или уменьшение значения символьной переменной. В арифметических вычислениях тип `char` преобразуется к типу `int`.

2.6.5 Логический тип

Логический (булевский) тип `boolean` предназначен для хранения логических значений. Переменные логического типа могут принимать только два значения: `true` (истина) и `false` (ложь). Описание переменных этого типа имеет следующий вид:

```
boolean log = true, flag;
```

2.6.6 Приведение типов

Часто программистам приходится присваивать переменным одного типа значения другого типа. Если типы совместимы, то осуществляется автоматическое преобразование типов. Например, всегда можно присвоить переменной типа `long` значение типа `int`, так как длина и диапазон допустимых значений `long` позволяет это сделать. Если типы несовместимы, то автоматическое преобразование не будет выполнено. В этом случае выполняют явное преобразование одного типа в другой.

Практически каждый простой тип можно преобразовать в любой другой, за исключением типа `boolean`. Операция преобразования одного типа данных в другой называется *приведением типов*.

На рис. 2.4 показаны допустимые преобразования числовых типов.

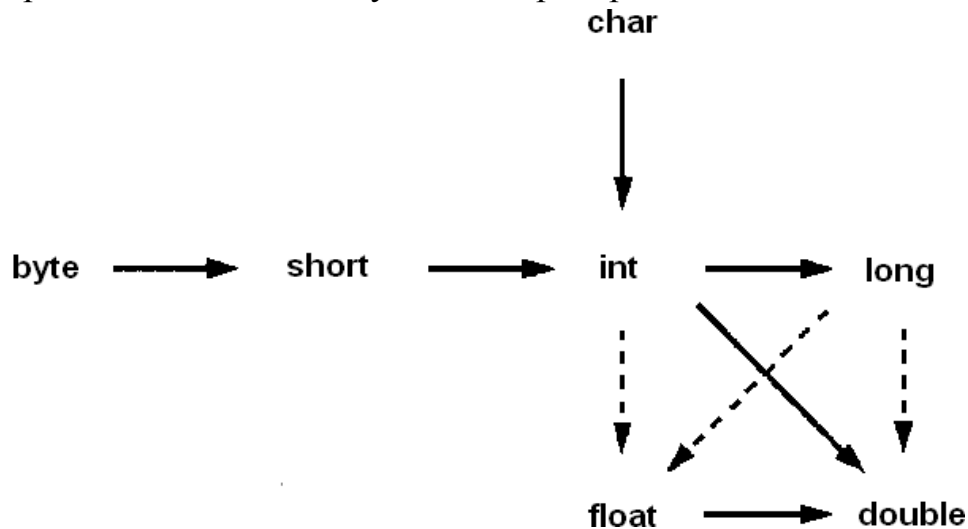


Рис.2.4 – Допустимые преобразования числовых типов

Шесть сплошных линий со стрелками обозначают преобразования, которые выполняются без потери информации. Три штрихованные линии, также со стрелками, означают преобразования, при которых может произойти потеря точности.

Автоматическое преобразование типа с расширением выполняется в случае, если оба типа совместимы и длина целевого типа больше длины исходного. Рассмотрим пример:

```
class Conversion {
    public static void main (String[] args) {
        byte a = 35;
        float b = 56.23f;
        b=a;
        System.out.println(b);
    }
}
```

После присваивания переменной `b` типа `float` значения переменной `a` типа `byte`, выполнится автоматическое преобразование типа с расширением, в результате чего старое значение `b` будет замещено на новое – 35.0.

Как видно из примера, числовые типы совместимы друг с другом. Однако не существует автоматических преобразований числовых типов в тип `char` или `boolean`. Типы `char` и `boolean` также не совместимы и между собой.

Явное преобразование типов применяется, когда необходимо выполнить преобразование между двумя несовместимыми типами.

Синтаксис явного преобразования типов имеет следующий вид:

(целевой_тип) значение

Здесь *целевой_тип* определяет тип, в который нужно преобразовать указанное значение.

Преобразование типа данных большей размерности к меньшей называется преобразованием с *сужением*, поскольку значение явно сужается, чтобы оно могло уместиться в целевом типе.

При присваивании значения с плавающей точкой переменной целого типа выполняется другой вид преобразования типа – *усечение*. Целые числа не содержат дробной части поэтому, когда вещественное значение присваивается переменной целого типа, дробная часть отбрасывается. Так, например, если в рассмотренном выше классе *Conversion* выполнить присваивание с явным приведением типов вида: `a=(byte)b`, то переменная *a* примет значение 56, так как дробная часть – 0.23 – отсекается.

Если размер целочисленной части слишком велик, чтобы уместиться в целевом целом типе, значение будет уменьшено до результата деления по модулю на диапазон целевого типа.

2.7 Операции над данными

Операция – это действие над одной или более (чаще всего двумя) величинами, называемыми *операндами*, приводящее к появлению новой величины – результата операции.

В зависимости от количества операндов операции делятся на унарные и бинарные. *Унарные* операции применяются к одному операнду. Символ унарной операции записывается перед операндом, например, `!log`. *Бинарные* операции применяются к двум операндам и в этом случае, символ операции записывается между операндами, например, `a*b`.

Все операции, выполняемые над данными, могут быть отнесены к одной из следующих четырех групп: арифметические операции, побитовые операции, операции сравнения и логические операции.

2.7.1 Арифметические операции

К арифметическим операциям относятся:

- сложение + ;
- вычитание – ;

- умножение * ;
- деление /;
- взятие остатка от деления (деление по модулю) % ;
- инкремент ++ ;
- декремент --.

Операнды арифметических операций должны иметь числовой тип. Допускается применение в арифметических операциях типа char, поскольку в Java этот тип, по сути, является разновидностью целого типа.

Операции сложение, вычитание, умножение и деление выполняются, как это обычно принято в математике, за исключением деления целых значений. В Java принято целочисленное деление. Это значит, что деление одного целого операнда на другой целый операнд в результате дает опять целое, например, 5/2 даст в результате 2, а не 2.5. Дробная часть отбрасывается и происходит усечение частного. Это очевидно, так как результат деления имеет такой же типа, что и операнды. Достаточно написать 5/2. или 5./2 и получим 2.5 как результат деления вещественных чисел.

Операция деления по модулю %, возвращает остаток деления и определяется следующим образом: $a \% b = a - (a / b) * b$. Причем $-a \% b = -(a \% b)$, но $a \% -b = a \% b$.

Таким образом, $11 \% 3 = 2$, $11 \% (-3) = 2$, $(-11) \% 3 = -2$. Операцию деления по модулю можно применять как к целым типам, так и к вещественным типам.

Задача 2.1. Определите все цифры целого положительного четырехзначного числа.

```
class Number{
    public static void main(String[] args){
        int d=1890;//заданное число
        int k1,k2,k3,k4;
        k1=d/1000;    //первая цифра
        k2=d/100%10;  //вторая цифра
        k3=d/10%10;   //третья цифра
        k4=d%10;      //четвертая цифра
        System.out.println("Первая цифра - "+k1);
        System.out.println("Вторая цифра - "+k2);
        System.out.println("Третья цифра - "+k3);
        System.out.println("Четвертая цифра - "+k4);
    }
}
```

Операции инкремент ++ и декремент -- соответственно увеличивают и уменьшают значение операнда на единицу. Таким образом, выражение $x++$ эквивалентно выражению $x=x+1$, а $x--$ – эквивалентно

$x=x-1$. Данные операции можно применять только к переменным. Нельзя применять их к константам или выражениям, следующие записи являются неверными: $++6$, $(x*y)++$, $++7+a$ (но $++a+7$ – верно).

Операции инкремента и декремента могут быть записаны как перед операндом (префиксная форма), так и после операнда (постфиксная форма). Разница в их выполнении проявляется только в выражениях.

В *префиксной форме* значение операнда увеличивается или уменьшается до извлечения значения для использования в выражении.

В *постфиксной форме* предыдущее значение извлекается для использования в выражении, и лишь после этого значение операнда изменяется.

Попробуем разобраться в том, как выполняются разные формы на примере операции инкремента.

<code>int a = 35;</code>	<code>a= 36</code>	В случае префиксной формы инкремента переменная <code>a</code> сначала увеличит свое значение на единицу, и станет равной 36, а потом это новое значение будет присвоено переменной <code>b</code> .
<code>int b = ++a;</code>	<code>b= 36</code>	
<code>int a = 35;</code>	<code>a= 36</code>	В случае постфиксной формы инкремента сначала переменной <code>b</code> будет присвоено значение <code>a</code> , и <code>b</code> станет равным 35, а потом переменная <code>a</code> увеличит свое значение на единицу.
<code>int b = a++;</code>	<code>b= 35</code>	

Таким образом, в случае префиксной формы переменная сначала изменяет на единицу свое значение, а потом ее новое значение участвует в вычислении выражения, а при постфиксной форме – сначала значение переменной участвует в вычислении выражения, а только потом, в самом конце, переменная изменяет свое значение на единицу.

2.7.2 Автоматическое повышение типа в выражениях

При вычислении выражений Java автоматически повышает тип каждого операнда `byte` или `short` до `int`. Благодаря этому решается проблема возможного выхода промежуточного значения выражения за пределы допустимого диапазона любого из операндов. Автоматическое повышение типа может приводить к неочевидным ошибкам во время компиляции. Рассмотрим пример:

```
byte b1 = 10, b2 = -10;
byte b3 = b1 + b2; // Ошибка! Значение типа int не
                  //может быть присвоено переменной типа byte
```


Во время вычисления выражения тип операндов был автоматически повышен до `int`, тип результата также был повышен до `int`. Таким образом, теперь результат выражения имеет тип `int`, который не может быть присвоен переменной типа `byte` без преобразования типа. Ошибку можно устранить с помощью явного приведения типов:

```
byte b1 = 10, b2 = -10;  
byte b3 = (byte) (b1 + b2); // b3=0
```

Сформулируем *правила повышения типа*:

- Тип всех значений `byte`, `short` и `char` повышается до `int`.
- Если один операнд имеет тип `long`, тип всего выражения повышается до `long`. Если один операнд имеет тип `float`, тип всего выражения повышается до `float`. Если любой из операндов имеет тип `double`, типом результата будет `double`.

2.7.3 Операции сравнения

В языке Java имеются шесть операций сравнения, которые определяют отношение одного операнда с другим (например, равенство и порядок следования):

- равно `==`;
- не равно `!=`;
- больше `>`;
- меньше `<`;
- больше или равно `>=`;
- меньше или равно `<=`.

Результат выполнения этих операторов – логическое значение `boolean`.

Выполнять операции равенства¹⁷ `==` и неравенства `!=` можно над значениями любых примитивных типов. Сравнение с помощью операций `>`, `<`, `>=`, `<=` можно выполнять только для целочисленных операндов, операндов с плавающей точкой или символьных операндов. Приведем несколько примеров использования операций сравнения с разными простыми типами.

```
boolean b1 = true;  
boolean b2 = false;  
boolean b3 = b1!=b2;  
char ch = 'c';  
boolean log = ch>'f';
```

В данном случае результат выполнения операции `b1!=b2`, который равен `true` сохраняется в переменной `b3`.
Результат выполнения операции `ch>'f'` равен `false`, так как при сравнении двух

¹⁷ Не путайте операцию сравнения на равенство `==` и оператор присваивания `=`.

```
double a=3.;
int b = 3;
boolean log = a>=b;
```

символов фактически сравниваются их коды 99 (символ 'c ') и 102 (символ 'f ').
В данном случае результат выполнения операции `a>=b` равен `true`, так как сравниваются одинаковые значения (`a=b=3`).

2.7.4 Логические операции

Логические операции, применимые к булевым значениям, приведены в табл. 2.5. Результатом выполнения данных операций будет тоже логическое значение `true` или `false`. Результаты выполнения логических операций с различными операндами приведены в табл. 2.6.

Таблица 2.5 – Логические операции*

Название операции	Оператор	Описание
Отрицание (NOT)	!	Отрицание меняет значение истинности.
Конъюнкция (AND)	&	Конъюнкция истинна, только если оба операнда истинны.
Дизъюнкция (OR)		Дизъюнкция ложна, только если оба операнда ложны.
Исключающее ИЛИ (XOR)	^	Исключающее ИЛИ истинно, только если значения операндов различны.

*Знак & называется амперсандом, ^ – каре.

Таблица 2.6 – Результаты выполнения логических операций

A	B	!A	A&B	A B	A^B
false	false	true	false	false	false
false	true	true	false	true	true
true	false	false	false	true	true
true	true	false	true	true	false

Кроме перечисленных четырех логических операций есть еще две логические операции сокращенного вычисления¹⁸:

¹⁸ Удвоенные знаки амперсанда и вертикальной черты следует записывать без пробелов.

- сокращенная конъюнкция (conditional-AND) `&&`;
- сокращенная дизъюнкция (conditional-OR) `||`.

Правый операнд сокращенных операций вычисляется только в том случае, если от него зависит результат операции, т.е. если левый операнд конъюнкции имеет значение `true`, или левый операнд дизъюнкции имеет значение `false`. Например, в следующем выражении применение сокращенной конъюнкции позволяет избежать деления на ноль, так как в случае равенства первого операнда нулю, второй операнд просто не будет вычисляться: `x!=0 && (y/x>0)`. Практически всегда в Java используются именно сокращенные логические операции.

Задача 2.2. Определите, встречается ли в записи положительного целого двузначного числа хотя бы одна нечетная цифра.

```
class Number{
    public static void main(String[] args){
        int N=47; // заданное число
        int k1=N/10; //первая цифра
        int k2=N%10; //вторая цифра
        boolean result=(k1%2!=0||k2%2!=0);
        System.out.println(result);
    }
}
```

2.7.5 Побитовые операции

В языке Java есть четыре побитовых (англ. bitwise) операций, которые изменяют значения отдельных битов операндов и могут применяться к целым типам: `long`, `int`, `short`, `char` и `byte`:

- дополнение (complement)¹⁹ `~`;
- побитовая конъюнкция (bitwise AND) `&`;
- побитовая дизъюнкция (bitwise OR) `|`;
- побитовое исключающее ИЛИ (bitwise XOR) `^`.

Они выполняются поразрядно, после того как оба операнда будут приведены к одному типу `int` или `long`, так же как и для арифметических операций. Операции над каждой парой битов выполняются согласно табл. 2.7.

Таблица 2.7 – Результаты выполнения побитовых операций

A	B	~A	A&B	A B	A^B
0	0	1	0	0	0

¹⁹ Знак `~` называется тильда.

1	0	0	0	1	1
0	1	1	0	1	1
1	1	0	1	1	0

Рассмотрим примеры выполнения побитовых операций. Пусть заданы две целые переменные $a=18$ и $b=-65$. Напомним, что двоичное представление отрицательных чисел в Java хранится в дополнительном коде. Перед выполнением операции будет выполнено повышение типа операндов до типа `int`, поэтому двоичное представление переменных будет состоять из 32-х разрядов:

$a = 0 \dots 00010010$, $b = 1 \dots 10111111$.

Побитовая операция	Результат операции	Двоичное представление результата
$\sim a$	-17	$1 \dots 11101111$
$\sim b$	64	$0 \dots 00100000$
$a \& b$	16	$0 \dots 00010000$
$a b$	-65	$1 \dots 10111111$
$a \wedge b$	-81	$1 \dots 10101111$

Напомним, что для декодирования отрицательного числа из дополнительного кода в десятичный, необходимо вначале инвертировать все биты, а затем добавить 1 к результату и изменить знак. Например, инвертирование значения -17, или 11101111 , приводит к значению 00010000 , или 16, после добавления 1 имеем 17, изменив знак, получаем результат -17.

2.7.6 Сдвиги

В языке Java есть три операции сдвига двоичных разрядов:

- сдвиг влево `<<`;
- сдвиг вправо `>>`;
- беззнаковый сдвиг вправо `>>>`.

Общая форма записи операций сдвига имеет следующий вид:

значение сдвиг количество

где количество определяет количество позиций сдвига значения значение, сдвиг – операция сдвига `<<`, `>>` или `>>>`.

Левый сдвиг `<<` смещает влево биты указанного значения на заданное количество позиций. При каждом сдвиге влево самый старший бит смещается за пределы допустимого диапазона и теряется, а справа дописывается нуль. Например, для $a=18$ и $b=-65$:

$a = 0 \dots 00010010$, $b = 1 \dots 10111111$.

Сдвиг	Результат операции	Двоичное представление результата
a<<1	36	0...00100100
b<<1	-130	1...01111110
a<<2	72	0...01001000
b<<2	-260	1...1011111100
a<<3	144	0...10010000

Заметьте, что сдвиг влево на n разрядов эквивалентен умножению числа на 2^n .

Правый сдвиг >>, называемый также арифметическим, смещает все биты значения вправо на указанное количество позиций, при этом слева распространяется старший бит, правые биты теряются. Для $a=18$ и $b=-65$:

$a = 0...00010010$, $b = 1...10111111$.

Сдвиг	Результат операции	Двоичное представление результата
a>>1	9	0...00001001
b>>1	-33	1...11011111
a>>2	4	0...00000100
b>>2	-17	1...11101111
a>>3	2	0...00000010

Как видно из примера, каждый раз при сдвиге значения вправо, оно делится на два и отбрасывает остаток. Эту особенность можно использовать для организации целочисленного деления на 2.

При выполнении сдвига вправо старшие биты, освобожденные в результате сдвига, заполняются предыдущим содержимым старшего бита. Этот эффект называется *дополнительным знаковым разрядом* и служит для сохранения знака отрицательных чисел при их сдвиге вправо. Если это является нежелательным, то используют операцию беззнакового сдвига.

Беззнаковый сдвиг >>> во всех случаях ставит слева на освободившиеся места нули, осуществляя логический сдвиг.

2.7.7 Условная операция

Язык Java содержит специальную троичную (с тремя операндами) условную операцию, которая имеет следующую общую форму:

выражение1 ? выражение2 : выражение3

где выражение1 – любое логическое выражение, приводящее к значению типа boolean. Если значение выражение1 – true, то вычисляется значение выражение2 и его значение будет результатом всей операции, выражение3 при этом не вычисляется. Если значение выражение1 – false, то вычисляется только значение выражение3 и оно будет результатом операции. И выражение2, и выражение3 должны возвращать значение одинакового (или совместимого) типа.

Условная операция очень удобна для организации небольших ветвлений. Обратите внимание, как просто можно записать условную операцию для нахождения минимального из двух значений:

```
min = (a < b) ? a : b;
```

Задача 2.3. Дано целое число *a*, определить, является ли оно четным или нечетным.

```
class Number{  
    public static void main(String[] args){  
        int N=47; // заданное число  
        System.out.println(N%2==0? "Четное" : "Нечетное");  
    }  
}
```

2.7.8 Оператор присваивания

В программировании под оператором принято понимать наименьшую автономную часть языка (также называемую инструкцией или командой) способную выполняться самостоятельно и несущую самостоятельный смысл. Программа обычно представляет собой последовательность операторов (команд, инструкций), которые заканчиваются символом точки с запятой (;).

Одним из простейших операторов является *оператор присваивания* (англ. assignment operator), который записывается знаком равенства =. Слева от оператора должна быть записана переменная, а справа выражение, совместимое с типом переменной. Например, $a=2*b+(c+6.35)$.

При выполнении оператора присваивания выражение, стоящее после знака равенства, вычисляется и приводится к типу переменной, стоящей слева от знака равенства. В результате переменная, стоящая слева, получает приведенное значение правой части, а старое ее значение теряется.

Оператор присваивания позволяет создавать цепочки присваиваний, как например, в следующем фрагменте программы:

```
int x, y, z;  
x = y = z = 1;
```

В результате всем трем переменным будет присвоено значение 1.

Кроме простого оператора присваивания в Java есть составные арифметические и побитовые операторы присваивания: `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `>>>=`. Правило записи всех составных операторов одинаково, любой оператор, имеющий форму:

переменная = переменная оператор выражение;
можно записать в следующем виде:
переменная оператор = выражение;

Это значит, что следующие записи в Java являются эквивалентными:

<code>x=x+5;</code>	<code>x+=5;</code>
<code>x=x*y;</code>	<code>x*=y;</code>
<code>x=x%3;</code>	<code>x%=3;</code>
<code>x=x<<2;</code>	<code>x<<=2;</code>

При использовании арифметических составных операторов присваивания перед присваиванием, при необходимости, автоматически производится приведение типа к типу переменной, которой присваивается выражение. Например,

```
byte b = 5;
int c = 350;
b = b + 25; // Ошибка!
b += 25; // Правильно
b += c; // Правильно
```

Как известно в Java в арифметических выражениях происходит повышение типа операндов до типа `int`, поэтому результат сложения `b+25` тоже будет иметь тип `int` и не может быть присвоен переменной `b` без явного приведения типа. Во втором и третьем случае перед присваиванием произойдет сужение результата сложения до типа `byte`. Причем в третьем случае сужение приведет к отбрасыванию старших битов результата сложения `b+c`. Если результат сложения равен 355 (в двоичном представлении – 101100011), то после сужения типа до `byte` получим 01100011 или 99.

Использование составных операторов присваивания позволяет уменьшить объем вводимого кода. Опытные программисты чаще используют именно такие операторы, так как они более эффективно реализуются в системе времени выполнения Java, чем эквивалентные длинные формы.

2.7.9 Приоритет операций

При использовании нескольких операций в одном выражении порядок их выполнения формируется компилятором на основании приоритетов операций. Перечень операций в порядке убывания их приоритетов приведен ниже. Операции на одной строке имеют одинаковый приоритет.

1. Постфиксные операции ++ и --.
2. Префиксные операции ++ и --, дополнение ~ и отрицание !
приведение типа (тип).
3. Умножение *, деление / и взятие остатка %.
4. Сложение + и вычитание -.
5. Сдвиги <<, >>, >>>.
6. Сравнения >, <, >=, <=.
7. Сравнения ==, !=.
8. Побитовая конъюнкция &.
9. Побитовое исключающее ИЛИ ^.
10. Побитовая дизъюнкция |.
11. Конъюнкция &&.
12. Дизъюнкция ||.
13. Условная операция ?:.
14. Присваивания =, +=, -=, *=, /=, %=, &=, ^=, |=, <<, >>, >>>.

2.7.10 Выражения

Выражения (англ. expressions) – это последовательность лексем и переменных, операций и вызовов методов, выполняемых в определенном порядке (по приоритету операций, с учётом скобок), дающая при вычислении некоторое значение.

При вычислении выражения выполняются следующие правила:

- Операции одного приоритета вычисляются слева направо²⁰.
- Левый операнд вычисляется раньше правого.
- Операнды полностью вычисляются перед выполнением операции.
- Перед выполнением составной операции присваивания значение левой части сохраняется для использования в правой части.

Порядок вычисления выражения всегда можно отрегулировать скобками. Круглые скобки () повышают приоритет заключенных в них операций. Кроме того, скобки облегчают понимание смысла выражения,

²⁰ Исключением является оператор присваивания, например, в следующей цепочке операторы присваивания вычисляются справа налево: $x = y = z$.

делают его более наглядным. При этом применение даже избыточных скобок в выражении не ведет к снижению производительности программы.

Особенности применения перечисленных правил рассмотрим на примерах.

Задача 2.4. Определите, чему будут равны переменные x , y и z после выполнения следующего фрагмента программы.

```
int x=12, y=-5, z;
    z=x+y;
    x%=y+3;
    y=--x;
    x=y+23%z++;
```

Таблица 2.8 – Пошаговое выполнение операторов (задача 2.4)

Оператор	Значения переменных		
	x	y	z
Начало	12	-5	0
$z=x+y$	12	-5	7
$x\%=y+3 \Rightarrow x=x\% (y+3)$	0	-5	7
$y=--x$	-1	-1	7
$x=y+23\%z++$	1	-1	8

Задача 2.5. Определите, чему будут равны переменные x , y и $b1$ после выполнения следующего фрагмента программы.

```
int x=0, y=1;
    boolean b1 = ++x==y--;
```

Таблица 2.9 – Пошаговое выполнение операторов (задача 2.5)

Оператор	Эквивалентный порядок выполнения операторов	Значения переменных		
		x	y	b1
Начало		0	1	
$++x==y++$	$x=x+1;$	1	1	
	$b1=(x==y);$	1	1	true
	$y=y-1;$	1	0	true

Задача 2.6. Определите, чему будут равны переменные $b1$ и $b2$ после выполнения следующего фрагмента программы.

```
int x=-8, y=3;
boolean b1 = x!=y^x+y==0;
int b2 = x*y<0||x==y?x+y:y-x;
```

Решение

Укажем явно приоритет выполнения операций в выражениях с помощью скобок.

```
x!=y^x+y==0 => (x!=y)^((x+y)==0)=> true^false  
=> b1 =true
```

```
x*y<0||x==y?x+y:y-x => (((x*y)<0)|| (x==y))?(x+y):(y-x)  
=>(true||false) ? -5 : 11 => b2 = -5
```

Ответ: b1 = true , b2 = -5

2.8 Методы класса Math

Отдельные математические функции и константы, применяемые в геометрии и алгебре, заданы в классе Math, который находится в пакете java.lang. Класс Math содержит методы, реализующие данные математические функции. Для использования методов в своих программах достаточно вызвать их по имени с указанием имени класса Math.

В классе Math заданы две константы:

```
double Math.E = 2.7182818284590452354;  
double Math.PI = 3.14159265358979323846;
```

2.8.1 Тригонометрические функции

Рассмотрим набор тригонометрических функций класса Math (табл.2.10). При их вызове в качестве входного параметра (аргумента записанного в круглых скобках метода) передается значение типа double – угол в радианах. Методы возвращают результат выполнения соответствующей тригонометрической функции тоже типа double.

Таблица 2.10 – Тригонометрические методы класса Math

Метод	Формула	Описание
<i>Прямые тригонометрические функции</i>		
sin(x)	sin(x)	Возвращает синус угла x, переданного в радианах
cos(x)	cos(x)	Возвращает косинус угла x, переданного в радианах
tan(x)	tg (x)	Возвращает тангенс угла x, переданного в радианах
<i>Обратные тригонометрические функции</i>		
asin(x)	arcsin(x)	Возвращает угол в радианах, синус которого равен x
acos(x)	arccos(x)	Возвращает угол в радианах, косинус

Метод	Формула	Описание
		которого равен x
<code>atan(x)</code>	$\arctg(x)$	Возвращает угол в радианах, тангенс которого равен x
<code>atan2(y, x)</code>		Возвращает угол в радианах, тангенс которого равен x/y
<i>Гиперболические функции</i>		
<code>sinh(x)</code>	$sh(x) = \frac{e^x - e^{-x}}{2}$	Возвращает гиперболический синус угла x , переданного в радианах.
<code>cosh(x)</code>	$ch(x) = \frac{e^x + e^{-x}}{2}$	Возвращает гиперболический косинус угла x , переданного в радианах.
<code>tanh(x)</code>	$th(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	Возвращает гиперболический тангенс угла x , переданного в радианах.
<i>Дополнительные функции</i>		
<code>toRadians(angle)</code>	$\alpha[\delta\alpha\ddot{\alpha}] = \alpha[^\circ] \cdot (\pi/180)$	Преобразует градусы в радианы. Угол, переданный аргументом <code>angle</code> , должен быть указан в градусах.
<code>toDegrees(angle)</code>	$\alpha[^\circ] = \alpha[\delta\alpha\ddot{\alpha}] \cdot (180 / \pi)$	Преобразует радианы в градусы. Угол, переданный аргументом <code>angle</code> , должен быть указан в радианах. Возвращается результат в градусах
<code>hypot(x, y)</code>	$\sqrt{x^2 + y^2}$	Возвращает длину гипотенузы по двум катетам x и y .

2.8.2 Экспоненциальные функции

Описание экспоненциальных (показательных) методов класса `Math` приведено в табл.2.11. Если в описании метода нет специального разъяснения, то по умолчанию тип входного параметра и результата выполнения метода – `double`.

Таблица 2.11 – Экспоненциальные методы класса `Math`

Метод	Формула	Описание
<code>exp(x)</code>	e^x	Возвращает экспоненту x .
<code>expm1(x)</code>	$e^x - 1$	Возвращает экспоненту $x-1$. При x близком к 0, дает более точные значения, чем <code>exp(x) - 1</code> .
<code>log(x)</code>	$\ln(x)$	Возвращает натуральный логарифм.
<code>log10(x)</code>	$\log_{10}(x)$	Возвращает логарифм по основанию 10 от x .
<code>log1p(x)</code>	$\ln(1 + x)$	Возвращает натуральный логарифм $x+1$.

Метод	Формула	Описание
<code>sqrt(x)</code>	$\sqrt{x}$	Возвращает квадратный корень из x .
<code>cbrt(x)</code>	$\sqrt[3]{x}$	Возвращает кубический корень из x .
<code>pow(x, y)</code>	x^y	Возвращает x в степени y .
<code>scalb(x, n)</code>	$x \cdot 2^n$	Возвращает значение $x \cdot 2^n$. Параметр n имеет тип <code>int</code> . Параметр x может быть <code>double</code> , тогда и результат выполнения метода будет <code>double</code> , либо <code>float</code> , тогда и результат будет иметь тип <code>float</code> .

2.8.3 Функции округления

Методы класса `Math`, предназначенные для выполнения различных операций округления, перечислены в табл. 2.12. По умолчанию тип их аргументов и возвращаемых значений – `double`.

Таблица 2.11 – Методы округления класса `Math`

Метод	Описание
<i>Методы нахождения модуля, знака, минимального и максимального числа</i>	
<code>abs(x)</code>	Возвращает абсолютное значение числа. Аргумент типа <code>int</code> , <code>long</code> , <code>float</code> или <code>double</code> . Результат того же типа, что аргумент.
<code>signum(x)</code>	Определяет знак числа. Возвращает 0, если x равен 0, 1 – если x больше 0 и -1 – если x меньше 0. Аргумент типа <code>float</code> или <code>double</code> . Результат того же типа, что и аргумент.
<code>copySign(x, s)</code>	Возвращает x с тем же знаком, что у s . Аргументы одного типа. Возможны типы: <code>float</code> и <code>double</code> . Результат того же типа, что и аргументы.
<code>min(x, y)</code>	Возвращает минимальное из двух чисел. Аргументы одного типа. Возможны типы: <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code> . Результат того же типа, что аргумент.
<code>max(x, y)</code>	Возвращает максимальное из двух чисел. Аргументы одного типа. Возможны типы: <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code> . Результат того же типа, что аргумент.
<i>Методы округления</i>	
<code>ceil(x)</code>	Возвращает наименьшее целое число, которое больше x

Метод	Описание
<code>floor(x)</code>	Возвращает наибольшее целое число, которое меньше или равно <code>x</code> .
<code>round(x)</code>	Возвращает ближайшее к <code>x</code> целое. Аргумент типа <code>float</code> или <code>double</code> . Результат типа <code>long</code> , если аргумент <code>double</code> , и типа <code>int</code> – если <code>float</code> . То же, что <code>(int) floor(x + 0.5)</code> .
<code>rint(x)</code>	Возвращает ближайшее целое к <code>x</code> .
<code>ulp(x)</code>	Расстояние до ближайшего большего чем аргумент значения того же типа ("дискретность" изменения чисел в формате с плавающей точкой вблизи данного значения). Можно использовать для получения точности результата. Аргумент типа <code>float</code> или <code>double</code> . Результат того же типа, что аргумент.
<code>nextUp(x)</code>	Возвращает следующее значение в положительном направлении от <code>x</code> . Аргумент типа <code>float</code> или <code>double</code> . Результат того же типа, что аргумент.
<code>nextAfter(x, d)</code>	Начиная со значения <code>x</code> возвращает следующее значение в направлении <code>d</code> . Если <code>x==d</code> , то возвращается <code>d</code> . Аргумент <code>d</code> имеет тип <code>double</code> . Аргумент <code>x</code> типа <code>float</code> или <code>double</code> . Результат того же типа, что и аргумент <code>x</code> .
<i>Дополнительные методы</i>	
<code>random()</code>	Возвращает псевдослучайное число в диапазоне от 0.0 до 1.0. При этом $0 \leq \text{Math.random}() < 1$. Если нужно получить псевдослучайное число в диапазоне от <code>x</code> до <code>y</code> , то можно использовать выражение: <code>(y-x) * Math.random() + x</code> .
<code>IEEEremainder(x, y)</code>	Возвращает остаток от целочисленного деления <code>x/y</code> .

2.9 Форматированный консольный ввод-вывод

При написании первой программы HelloWorld мы уже использовали вывод данных в консольном приложении, когда печатали результат на экран с помощью метода `System.out.println()`. Напомним, что наряду с методом `println()` существует также очень похожий метод `print()`. Единственное его отличие заключается в том, что он не осуществляет перевода строки. Методы `print()` и `println()` достаточно удобны и

позволяют выводить константы и переменные в самых различных сочетаниях, объединяя их с помощью операции сцепления:

```
int min=10, max=100;
System.out.println("Max=" + max + " Min=" + min);
```

В примере выводятся строковые литералы и значения переменных `max` и `min`. Переменные `max` и `min` сначала преобразуются из целого типа в строчный эквивалент и затем сцепляются со строками. В методах `print()` и `println()` можно использовать арифметические операции. В этом случае выражение будет вычислено и результат выведен на консоль.

Будьте внимательны при использовании операции сложения. У сцепления строк приоритет выше, чем у сложения, поэтому, записав:

```
System.out.println("max+min=" + max + min);
```

получим на экране строку `"max+min=10010"`. Но, записав:

```
System.out.println(max + min + "=max+min");
```

получим строку `"110=max+min"`, поскольку действия выполняются слева направо. Если нужно сначала вычислить сумму, а потом выполнить операцию сцепления, то используйте скобки:

```
System.out.println("max+min=" + (max + min));
```

получим `"max+min=110"`.

Попробуем вывести вещественное значение. Например, следующее выражение:

```
double x=3.5;
System.out.println("F=" +
    (Math.sin(x)+Math.cos(x))/Math.cbrt(x));
```

получим на экране: `F=-0.8478196600874199`.

Как видно результат выполнения выражения был выведен на экран в неудобном формате. Метод `println()` не позволяет управлять видом строкового представления, задавать количество знаков после запятой, выравнивать строку и др. Для форматированного вывода следует использовать метод `printf()` и спецификаторы формата, позволяющие программисту задавать отображение чисел и строк практически в любом виде.

2.9.1 Основы форматированного вывода данных разных типов

Для управления видом строкового представления может быть использован метод `System.out.printf()`. Он принимает в качестве аргументов форматную строку и переменное количество выражений и преобразует их в строку в формате, определенном форматной строкой. Например, `System.out.printf("x=%d y=%f", 38, 10.787);`;

Результат: `x=38 y=10,787000`

В примере в двойных кавычках записана форматная строка, а после запятой два аргумента – числовые значения, которые должны быть выведены – 38 и 10.787. В форматной строке указаны *спецификаторы формата*, определяющие способ, в соответствии с которым должны отображаться последующие аргументы.

Спецификатор формата начинается со знака процента с последующим спецификатором преобразования. Все спецификаторы преобразования формата состоят из единственного символа. Например, спецификатор формата для числа с плавающей точкой выглядит как `%f`. В общем случае должно быть столько аргументов, сколько есть спецификаторов формата, и соответствие аргументов устанавливается слева направо. В примере спецификаторы формата `%d` и `%f` замещаются аргументами, следующими за строкой формата. То есть `%d` заменяется на 38, а `%f` – на 10.787. Все остальные символы выводятся так, как они записаны в форматной строке. Ниже в табл. 2.12 приведены основные, часто используемые спецификаторы формата. Некоторые из них имеют и заглавную, и строчную формы. Единственное отличие этих форм состоит в том, что при использовании заглавной формы, буквы отображаются в верхнем регистре.

Таблица 2.12 – Спецификаторы формата

Спецификатор формата	Тип элемента данных
<code>%b</code> или <code>%B</code>	Логическое (булево) значение
<code>%c</code>	Одиночный символ
<code>%d</code>	Целое десятичное число
<code>%e</code> или <code>%E</code>	Число с плавающей точкой в экспоненциальной форме
<code>%f</code>	Десятичное число с плавающей точкой
<code>%g</code> или <code>%G</code>	Использует либо <code>%e</code> либо <code>%f</code> , в зависимости от такого с каким спецификатором представление числа короче.
<code>%o</code>	Восьмеричное число

<code>%n</code>	Вставляет символ перевода строки
<code>%s</code> или <code>%S</code>	Строка
<code>%t</code> или <code>%T</code>	Время и дата. Спецификатор требует применения специальных суффиксов.
<code>%x</code> или <code>%X</code>	Шестнадцатеричное число
<code>%%</code>	Вставляет символ %

Особый интерес представляет форматирование чисел. Для форматирования целых чисел применяют спецификатор формата `%d`. Чтобы форматировать значения с плавающей точкой применяют спецификатор `%f`. Кроме данного спецификатора можно использовать спецификатор `%e`, он представляет числа с плавающей точкой в экспоненциальной форме, состоящей из мантиссы и экспоненты и имеющей следующий вид: `x.ddddde+/-yy`. Например:

```
System.out.printf("x=%e", 10.787);
```

Результат: `x=1,078700e+01`

Спецификатор формата `%g` позволяет выводить число в самой короткой записи без потери точности, т.е. если заданное число имеет более короткую запись в экспоненциальной форме, то оно и выводится в этой форме.

Целые числа можно также выводить в восьмеричном или шестнадцатеричном формате, используя соответственно спецификаторы `%o` или `%x`. Например, `System.out.printf("x=%o y=%x", 98, 98);`

Результат: `x=142 y=62`.

При выводе целых чисел и символов с использованием спецификаторов формата аргумент занимает столько позиций, сколько требуется для его представления (12 – две позиции, -156 – четыре позиции и т.д.). Но можно и явно указывать количество позиций для вывода аргумента. Для целых чисел, символьных и строковых данных может быть указано только одно число. Например, форматная строка `"%4d"` обозначает, что число должно занимать 4 позиции. Если число занимает меньше 4 цифр, то число выравнивается вправо или, что является одним и тем же, дополняется пробелами слева. Например, число -5 будет в этом случае записано так: `_5`.

Если требуется изменить направление выравнивания, то перед числом в форматном коде следует записать знак минус. Так, форматный код `"%-4d"` обеспечит вывод числа -5 следующим образом: `-5_`. В этом случае выделены также 4 позиции, но 2 позиции после числа заняты пробелами, и в них нельзя разместить другую информацию.

Целое число, помещенное между символом % и кодом преобразования формата, выступает в качестве *спецификатора минимальной ширины*, который дополняет вывод пробелами, чтобы обеспечить заданную минимальную длину. Если строка или число получаются длиннее этого заданного минимума, они будут напечатаны полностью. По умолчанию дополнение осуществляется пробелами. Если следует дополнить нулями, необходимо поместить 0 перед спецификатором ширины поля. Например, в предыдущем случае при выводе числа -5 в формате "%04d" получим: -005. Число будет дополнено нулями так, чтобы его общая ширина была равна четырем.

Спецификаторы формата для чисел с плавающей точкой и строк могут содержать два числа, разделенные точкой. Например, "%6.3f". Первое число 6 обозначает общую ширину поля, занимаемую выводимым числом, а второе число 3, называемое *спецификатором точности*, обозначает количество разрядов в дробной части (для чисел с плавающей точкой в экспоненциальной форме второе число обозначает точность представления). По умолчанию точность чисел с плавающей точкой составляет 6 знаков после запятой.

Спецификатор точности может быть применен к спецификаторам формата %f, %e, %g и %s. Он следует за спецификатором минимальной ширины поля (если таковой имеется) и состоит из точки с последующим целым числом. Рассмотрим следующий пример:

```
System.out.printf("x=%4.2e y=%4.2f",10.787,10.787);
```

Результат: x=1,08e+01 y=10,79

Как видно из примера в качестве разделителя целой и дробной частей вещественных чисел используется запятая. В зависимости от текущей локали,²¹ при выводе чисел с плавающей запятой может использоваться как запятая, так и точка. Поведение метода printf() в отношении разделителя определяется используемой локалью.

В применении к строкам спецификатор точности задает максимальную ширину поля. Например, "%5.7s" отображает строку длиной минимум в пять символов и не превышающей семь символов. Если строка длиннее максимальной ширины, конечные символы будут усечены. Пример форматирования строки:

```
System.out.printf("%8.4s", "Программа HelloWorld");
```

Результат: ____Прог

²¹ Локаль ([англ.](#) locale) – это набор параметров, включающий [набор символов](#), [язык](#) пользователя, [страну](#), [часовой пояс](#), а также другие предустановки.

2.9.2 Специальные флаги формата

В спецификаторе формата за знаком % могут быть записаны так называемые *флаги формата*. Некоторые из них были кратко рассмотрены ранее. Перечень флагов, которые могут быть использованы при форматировании данных, приведен в табл. 2.13.

Рассмотрим кратко на примерах неупомянутые ранее флаги.

Флаг + добавляют, чтобы заставить отображать знак + перед положительными числовыми значениями. Например:

```
System.out.printf("%+d", 250);
```

Результат:+250

Таблица 2.13 – Флаги формата

Флаг	Назначение
-	Выравнивание влево
#	Альтернативный формат преобразования
0	Дополнение нулями вместо пробелов при выводе
пробел	Добавление одного пробела перед положительными числами
+	Добавления знака "+" перед положительными числами
,	Добавление группового разделителя
(	Вывод отрицательных чисел в круглых скобках

Флаг (используют для вывода отрицательных числовых значений в скобках вместо добавления ведущего минуса. Например:

```
System.out.printf("(%d",-250);
```

Результат:(250)

Флаг пробела удобно использовать в том случае, когда необходимо вывести в столбец положительные и отрицательные числа. Благодаря ведущему пробелу у положительных чисел, обеспечивается ровное расположение разрядов в столбце. Например:

```
System.out.printf("% d%n",-250);
```

```
System.out.printf("% d%n",250);
```

```
System.out.printf("% d%n",-250);
```

```
System.out.printf("% d%n",250);
```

Результат:

-250
250
-250
250

Флаг `,` (запятая) добавляет разделители групп и удобен при отображении больших чисел. В зависимости от установленной локали в качестве разделителя групп может использоваться запятая либо пробел. Например:

```
System.out.printf("%.2f",12587432.5681);
```

Результат: 12 587 432.57

Флаг²² `#`, добавленный к спецификатору формата `%x`, позволяет вывести шестнадцатеричное число с префиксом `0x`. Если флаг `#` добавить к спецификатору `%o`, то восьмеричное число будет выведено с ведущим нулем. Например:

```
System.out.printf("%#x %#o",255,255);
```

Результат: 0xff 0377

Кроме перечисленных флагов формата для управления выводом можно использовать *индекс аргумента*, явно указывающий номер аргумент, к которому должен применяться конкретный спецификатор формата. Обычно порядок аргументов и спецификаторов формата совпадает – слева направо. Однако, можно изменить этот порядок, используя индекс аргумента, который имеет следующий вид: `n$`, где `n` – номер аргумента, к которому нужно применить спецификатор формата. Например:

```
System.out.printf("%2$d>%1$d",25,255);
```

Результат: 255>25

В примере изменен привычный порядок использования аргументов. Сначала выводится второй, а потом – первый аргумент.

Индексирование аргументов также позволяет повторно использовать аргумент, не указывая его дважды. Например:

```
System.out.printf ("%d в 16-ричном формате - %1$#x",  
255) ;
```

Результат: **255 в 16-ричном формате - 0xff**

Существует удобное сокращение, называемое *относительным индексом* `<`, позволяющее повторно использовать аргументы, соответствующие предшествующему спецификатору формата. Например, следующий вызов метода **printf()** вернет тот же результат, что и предыдущий пример.

²² Знак называется октоторп.

```
System.out.printf ("%d в 16-ричном формате - %#x", 255);
```

2.9.3 Консольный ввод с помощью класса Scanner

Для ввода данных с консоли в Java используются специальные базовые классы для ввода/вывода, расположенные в пакете `java.io` (`InputStreamReader` и `BufferedReader`). Для того чтобы осмысленно научиться их использовать, нужно знать основные понятия объектно-ориентированного программирования, которые будут излагаться позже. Так как ввод данных с консоли нужно уметь делать уже сейчас, то рассмотрим простой класс `Scanner`, который позволяет легко осуществить считывание примитивных типов данных и строк. Этот класс располагается в пакете `java.util`. Таким образом, чтобы его можно было использовать, необходимо в начале программы импортировать данный пакет, то есть написать `import java.util.Scanner;`

Далее по тексту программы, в том месте, где необходим ввод, нужно записать:

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

Если забежать вперед, то эта строка создает объект класса `Scanner`, который читает из стандартного потока ввода (по умолчанию с клавиатуры – `System.in`). Объект назван `scanner`. Но это идентификатор, имя можно дать любое. Например,

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

Эта строка записывается в программе только один раз, даже если нужно считывать данные несколько раз! Она просто подготавливает (настраивает) класс для ввода, а дальше нужно использовать один из методов `nextXXX()` для считывания примитивных типов и строк: `nextBoolean()`, `nextFloat()`, `nextInt()` и т.д. Список методов `next()` для считывания данных примитивных типов приведен в табл.2.14.

Таблица 2.14 – Методы next класса Scanner

Метод	Описание
<code>next()</code>	Возвращает следующую лексему любого типа из входного источника как строковое значение.
<code>nextLine()</code>	Возвращает следующую строку ввода
<code>nextBoolean()</code>	Возвращает следующую лексему как значение типа <code>boolean</code>
<code>nextByte()</code>	Возвращает следующую лексему как значение типа <code>byte</code> по основанию 10.
<code>nextByte(int n)</code>	Возвращает следующую лексему как значение типа <code>byte</code> по основанию <code>n</code>
<code>nextShort()</code>	Возвращает следующую лексему как значение

	типа short по основанию 10.
nextShort(int n)	Возвращает следующую лексему как значение типа short по основанию n
nextInt()	Возвращает следующую лексему как значение типа int по основанию 10.
nextInt(int n)	Возвращает следующую лексему как значение типа int по основанию n
nextLong()	Возвращает следующую лексему как значение типа long по основанию 10.
nextLong(int n)	Возвращает следующую лексему как значение типа long по основанию n
nextFloat()	Возвращает следующую лексему как значение типа float
nextDouble()	Возвращает следующую лексему как значение типа double

Метод next**xxx**() используется столько раз, сколько раз нужно ввести данные. Метод вводит отдельные слова (лексемы, числовые значения). Разделителем по умолчанию является пробел. Например, так можно ввести целое число:

```
int i = scanner.nextInt();
или если объект класса Scanner назван s, то
int i = s.nextInt();
```

Запишем фрагмент класса, в котором нужно ввести сначала целое число, а потом вещественное.

```
import java.util.Scanner;
public class Example_Scanner
{
    public static void main (String[] args){
        Scanner scan = new Scanner (System.in);
        int a = scan.nextInt();
        double b = scan.nextDouble();
    }
}
```

При вводе вещественных чисел будьте внимательны с разделителем. Иногда (это зависит от текущей локали) следует использовать в качестве разделителя запятую, а не точку.

Для приведенной программы строка ввода на экране может иметь следующий вид: 5 6,7.

Можно при вводе вставлять строки-подсказки, чтобы пользователь знал, что нужно вводить. Модифицируем предыдущую программу:

```
import java.util.Scanner;
```

```

public class Example_Scanner
{
    public static void main (String[] args){
        Scanner scan = new Scanner (System.in);
        System.out.println("Введите целое число:");
        int a = scan.nextInt();
        System.out.println("Введите вещественное число:");
        double b = scan.nextDouble();
    }
}

```

На экране это будет выглядеть следующим образом:

```

Введите целое число:
5
Введите вещественное число:
6,7

```

Класс Scanner определяет, где начинаются и заканчиваются лексемы, на основе набора разделителей. По умолчанию разделителями являются пробелы. Однако можно изменить разделители, вызвав метод `useDelimiter()`. В качестве входного параметра в метод нужно передать строку со списком символов, которые в дальнейшем будут восприниматься классом Scanner как разделители.

Ниже показана программа, которая читает список чисел, разделенных запятыми.

```

import java.util.*;
public class Example_Scanner
{
    public static void main (String[] args){
        Scanner scan = new Scanner (System.in);
        scan.useDelimiter(",");
        int a = scan.nextInt();
        int b = scan.nextInt();
        int c = scan.nextInt();
        System.out.printf("Сумма введенных чисел = %d", a+b+c);
    }
}

```

Задача 2.7. Рассчитайте $y = (-1)^n (2n + 1) / 2$, где n – произвольное целое число, которое вводится с клавиатуры;

Рассчитайте

$$z_1 = \frac{1}{2} \sin^2(3\pi - 2\alpha) \cos^2(5\pi + 2\alpha)$$

$$z = \frac{1}{2} - \frac{\sin^2 \pi - 8\alpha}{2}$$

$$z = \frac{1}{4} - \frac{1}{4} \left(\frac{2}{2} \right)$$

```

import java.util.Scanner;
class Calculation{
    public static void main (String[] args){
        Scanner scan = new Scanner (System.in);
    }
}

```

```

System.out.println("Введите целое число n:");
int n = scan.nextInt();
double y=Math.pow(-1,n)*(2*n+1)/2.;
System.out.println("Введите вещественное число a:");
double a = scan.nextDouble();
double z1=2*Math.pow(Math.sin(3*Math.PI-2*a),2)*
           Math.pow(Math.cos(5*Math.PI+2*a),2);
double z2=1/4.-1/4.*Math.sin(5/2.*Math.PI-8*a);
System.out.printf("y = %5.2f%n",y);
System.out.printf("z1 =%5.2f z2 =%5.2f",z1,z2);
    }}

```

Результат:

Введите целое число n:

3

Введите вещественное число a:

5,2

y = -3,50

z1 = 0,43 z2 = 0,43

Упражнения к главе 2

1. Определите, чему будут равны переменные x, y и z после выполнения следующего фрагмента программы:

```

int x=-2, y=2, z=y+x++;
    x*=y+z; y+=x+2; x=--y+5%3;

```

Ответ: x = 3 y = 1

2. Определите, что будет выведено на экран после выполнения следующего фрагмента программы:

```

int x=-1, y=5;
System.out.print("LOG= \n"+(x-y>0^++x!=0));
System.out.print(" Z= "+(y>x|x==0?x+y:y-x));

```

Ответ: LOG=
false Z= 5

3. Дана целая переменная a. Не используя никаких функций и никаких операций, кроме умножения, получить a^8 за три операции; a^{10} и a^{16} за четыре операции.

4. С помощью операций сдвига и сравнений выразите признак того, что беззнаковое целое x делится на 4 без остатка.

5. Даны целые положительные переменные a и b. Напишите булево выражение, значением которого является true тогда и только тогда, когда:

- целое a делится на b без остатка;
- целые значения a и b имеют одинаковую четность.

6. Даны целые переменные a и b ($a \neq b$). Используя условную операцию присвойте целой переменной c большее из двух значений a и b .
7. Дана целая переменная a . Напишите арифметическое выражение, значением которого является:
- значение последней цифры в десятичном представлении a ;
 - сумма значений десятичных цифр двузначного a ;
 - сумма значений десятичных цифр трехзначного a .
8. Напишите выражение, задающее вычисление следующего значения:
 $y = (-1)^{2n+1}(2n + 1) / 2$, где n – случайное целое число из диапазона $[-5 \div 25]$.
9. Даны целые переменные с положительными значениями a, b, c . Напишите булево выражение, значением которого является `true` тогда и только тогда, когда:
- a, b, c имеют одинаковые значения;
 - a, b, c задают стороны треугольника;
 - a, b, c задают стороны прямоугольного треугольника;
 - a, b, c задают стороны остроугольного треугольника;
 - a, b, c задают стороны равнобедренного треугольника;
 - a, b, c задают стороны разностороннего треугольника;
 - a, b, c задают стороны параллелограмма.
10. Напишите программу, определяющую может ли окружность радиуса R пройти в прямоугольное отверстие со сторонами a, b .
11. Напишите программу, определяющую угол (в градусах) между положением часовой стрелки в начале суток и ее положением в h часов, m минут, s секунд ($0 \leq h \leq 11$, $0 \leq (m, s) \leq 59$).
12. Напишите программу, определяющую больше ли площадь боковой поверхности цилиндра площади круга, являющегося основанием цилиндра. Радиус круга R , высота цилиндра h .
13. Напишите программу, определяющую для произвольных вещественных чисел a, b, c имеет ли квадратное уравнение хотя бы один действительный корень.
14. Напишите программу, которая выводит следующее изображение:

```

* * ***** * * *****
* * * * * * * * *
***** ***** * * * * *
* * * * * * * * *
* * ***** ***** *****

```

Вместо символа `*` предусмотреть вывод любого символа введенного пользователем.

15. С помощью методов класса `Math` запишите выражение, которое по вещественной переменной x задает вычисление следующих значений:

$$z_1 = \frac{x^2 + 2x - 3 + (x + 1)\sqrt{x^2 - 9}}{x^2 - 2x - 3 + (x - 1)\sqrt{x^2 - 9}}$$

$$z_2 = \sqrt{\frac{x + 3}{x - 3}}$$

3 УПРАВЛЯЮЩИЕ ОПЕРАТОРЫ

Для реализации основных алгоритмических структур используются *управляющие операторы*. В Java имеется большой набор управляющих операторов, который можно разделить на следующие категории: операторы выбора, операторы цикла и операторы перехода. Всякий оператор завершается точкой с запятой. Один из простых операторов – оператор присваивания уже был рассмотрен нами ранее. К оператором можно отнести и некоторые операции, так поставив точку с запятой после операций инкремента или декремента, получим оператор – отдельную команду, выполняющуюся самостоятельно.

3.1 Операторы выбора

Операторы выбора позволяют реализовать в программе алгоритмическую структуру ветвления: выбрать ту или иную последовательность выполнения команд (ветвь) в зависимости от истинности определенного условия. В языке Java поддерживаются два оператора выбора: `if` и `switch`.

3.1.1 Условный оператор `if`

Условный оператор `if` используют для направления выполнения программы по одной из двух ветвей в зависимости от истинности некоторого условия. Общая форма условного оператора выглядит следующим образом:

```
if (условие) оператор1; [else оператор2;]
```

Здесь `if` и `else` – зарезервированные служебные слова. Условие – произвольное логическое выражение. Оператор1 и оператор2 – это любой одиночный оператор или составной оператор, заключенный в фигурные скобки (блок).

Выполнение условного оператора начинается с вычисления значения булевого выражения в круглых скобках. Если условие истинно, то выполняется оператор1 и управление передается оператору, следующему за условным оператором (оператор2 пропускается). В противном случае, если условие ложно, то оператор1 пропускается, а выполняется оператор2 (если он существует) и на этом действие условного оператора считается завершенным. Ни в одной из ситуаций не будет выполнено оба оператора.

Обратите внимание, что в общей форме записи условного оператора фраза `else оператор2;` является необязательной и может

отсутствовать. В этом случае условный оператор будет иметь сокращенную форму. В сокращенной форме оператора `if` если условие является истинным, то выполняется оператор1, если ложным – не выполняется и на этом действие условного оператора считается завершенным. Блок-схемы полной и сокращенной форм условного оператора представлены на рис.3.1.

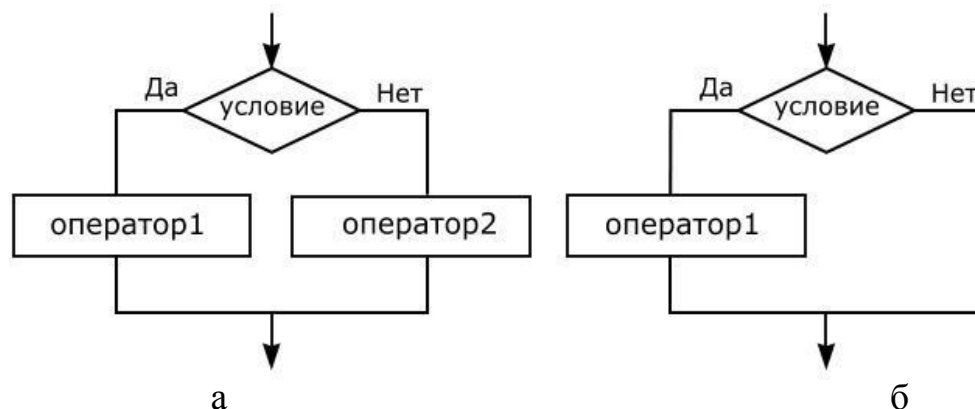


Рис. 3.1 Блок-схема условного оператора: полная форма (а); сокращенная форма (б)

Приведем простой пример использования условного оператора. Выберем из двух неравных целых чисел `a` и `b` наибольшее:

```
if (a>b) System.out.println("Max = "+a);
    else System.out.println("Max = "+b);
```

В примере, если значение переменной `a` больше значения переменной `b`, то на экран будет выведена переменная `a`, в противном случае – значение переменной `b`. В условии оператора `if` используется операция сравнения. Однако условие может состоять из нескольких логических операций или просто из отдельной булевой переменной. Например, как в следующем примере:

```
boolean b=true;
int a=0;
if (b) a++;else a--;
if (a>=-1&&a<=1)
    System.out.println("a принадлежит отрезку [-1;1]");
```

Как видно из общей формы записи условного оператора, за ключевым словом `if` или `else` непосредственно может следовать только один оператор. Если нужно включить больше операторов, их следует объединить в блок.

Блок – это произвольное количество простых операторов языка Java, заключенных в фигурные скобки с целью использования их как один оператор в тех местах, где по правилам языка можно использовать только один оператор. Блоки операторов часто используются для ограничения области действия переменных и просто для улучшения читаемости текста программы.

```
if (условие)
{ оператор;
  оператор;
  . . .
  оператор;
}
else
{ оператор;
  оператор;
  . . .
  оператор;
}
```

Тело **if** из нескольких операторов, заключенных в блок

Тело **else** из нескольких операторов, заключенных в блок

Задача 3.1. Даны целые числа a , b , c . Если $a \leq b \leq c$, то все числа заменить их квадратами, в противном случае сменить знак каждого числа.

```
import java.util.Scanner;
class Ex_3_1 {
public static void main (String[] args){
    Scanner s = new Scanner(System.in);
    System.out.println("Введите целое число a ->");
    int a = s.nextInt();
    System.out.println("Введите целое число b ->");
    int b = s.nextInt();
    System.out.println("Введите целое число c ->");
    int c = s.nextInt();
    if (a<=b&&b<=c) {a*=a; b*=b; c*=c;}
    else {a*=-1; b*=-1; c*=-1;}
    System.out.printf("a=%d b=%d c=%d",a,b,c);
}}
```

Результат:

```
Введите целое число a ->
5
Введите целое число b ->
7
Введите целое число c ->
9
a=25 b=49 c=81
```

3.1.2 Вложенный оператор if

Оператор if является вложенным, если он находится внутри другого оператора if или else. Рассмотрим общую форму записи и блок-схему вложенного условного оператора (рис.3.2). Здесь один оператор if вложен в ветвь else другого условного оператора:

```
if (условие1) оператор1;  
    else if (условие2) оператор2; else оператор3;
```

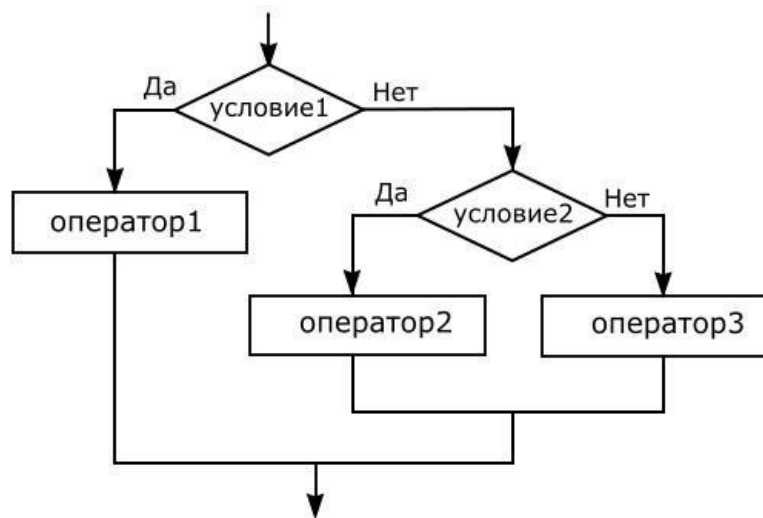


Рис.3.2 – Блок-схема вложенного условного оператора

Работу вложенного условного оператора рассмотрим на примере решения задачи 3.2.

Задача 3.2. Для вещественного числа x вычислить:

$$y = \begin{cases} x^2 & \text{if } x \leq 0, \\ x^2 - x & \text{if } 0 < x \leq 1, \\ x^2 - \sin \pi x & \text{if } x > 1. \end{cases}$$

```
import java.util.Scanner;  
class Ex_3_2 {  
    public static void main (String[] args) {  
        Scanner s = new Scanner(System.in);  
        System.out.println("Введите x ->");  
        double x = s.nextDouble();  
        double y;  
        if (x<=0) y=x*x;  
            else if (x>0&&x<=1) y=x*x-x;  
                else y=x*x-Math.sin(Math.PI*x);  
        System.out.printf("y=%8.3f", y);  
    }  
}
```

Если используется несколько операторов `if`, то `else` всегда ассоциируется с ближайшим `if`. Например:

```
int a = 0, b = 10;
if (a!=0) if (a==0) b = 0; else b = 1;
```

В данном фрагменте кода ветвь `else` относится к ближайшему слева условию `if`, не имеющему своей ветви `else`, поэтому переменная `b` останется равной 10. Изменить этот порядок можно с помощью блока, например:

```
if (a!=0) {if (a==0) b = 0;} else b = 1;
```

При истинности условия `a!=0` будет выполнен код в блоке, в противном случае – оператор присваивания в ветви `else`, которая в данном случае ассоциируется с первым `if`. Вообще не стоит увлекаться сложными вложенными условными операторами и по возможности использовать блоки для явного указания вложенности операторов.

Рассмотрим задачу 3.3, демонстрирующую использование вложенного условного оператора. На рис.3.3 изображена блок-схема алгоритма решения данной задачи.

Задача 3.3 Построить алгоритм определения квадранта декартовой системы, в котором находится точка с координатами x, y .

```
import java.util.Scanner;
class Ex_3_3 {
    public static void main (String[] args){
        Scanner s = new Scanner(System.in);
        System.out.println("Введите x ->");
        double x = s.nextDouble();
        System.out.println("Введите y ->");
        double y = s.nextDouble();
        int n=1;
        if (x>0&&y<0) n=4;
            else if (x<0&&y<0) n=3;
                else if (x<0&&y>0) n=2;
        System.out.printf("Номер квадранта - %d",n); }}
```

Правильность алгоритма можно проверить путем заполнения трассировочной таблицы. В табл. 3.1 приведена трассировка алгоритма для исходных значений $x=-5.0, y=10.0$

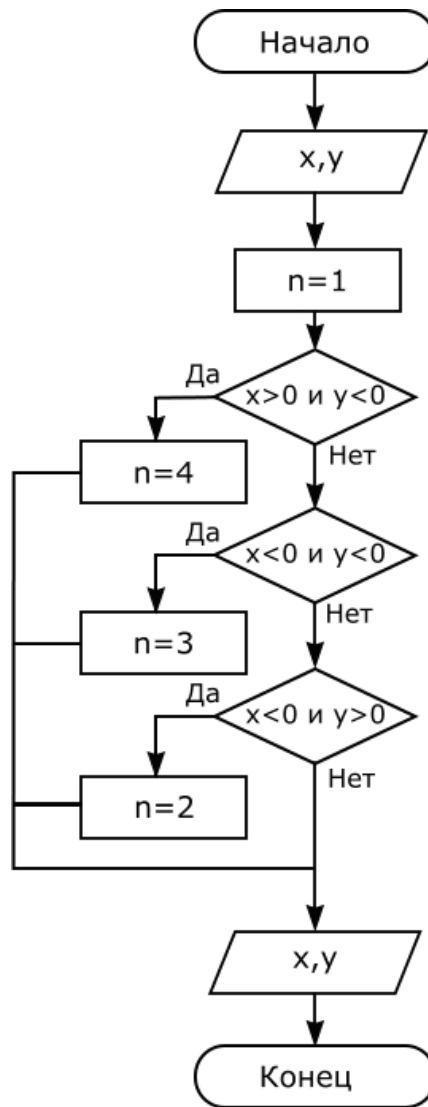


Рис.3.3 – Блок-схема алгоритма решения задачи 3.3

Таблица 3.1 – Трассировка алгоритма задачи 3.3

Шаг	Команда	x	y	n	Проверка условия
1	<code>double x= s.nextDouble();</code>	-5	-	-	
2	<code>double y = s.nextDouble();</code>	-5	10	-	
3	<code>int n=1;</code>	-5	10	1	
4	<code>x>0&&y<0</code>	-5	10	1	-5>0 и 10<0, false
5	<code>x<0&&y<0</code>	-5	10	1	-5<0 и 10<0, false
6	<code>x<0&&y>0</code>	-5	10	1	-5<0 и 10>0, true
7	<code>n=2</code>	-5	10	2	
8	<code>System.out.printf("Номер квадранта - %d", n);</code>	-5	10	<u>2</u>	

3.1.2 Оператор выбора switch

Оператор выбора `switch` организует разветвление по нескольким направлениям. Каждая ветвь отмечается константой или константным выражением и выбирается, если значение определенного выражения совпадет с этой константой. Общая форма записи оператора `switch` имеет вид:

```
switch (выражение){  
  [case константное-выражение1]:[список-оператор1]  
  [case константное-выражение2]:[список-оператор2]  
  . . .  
  [case константное-выражениеN]:[список-операторN]  
  [default: [список-операторDef]]  
}
```

Стоящее в скобках выражение может быть целого типа: `byte`, `short`, `int`, но не `long`; символьного типа `char`; строкового типа `String` или иметь тип перечисления. Каждое значение константного выражения, определенного в операторах `case`, должно быть уникальным константным выражением. Дублирование значений `case` не допускается. Тип каждого значения должен быть совместим с типом выражения.

Оператор `switch` работает следующим образом. Сначала вычисляется выражение. Если оно совпадает с одной из констант, то выполняется список операторов, отмеченный этой константой. Затем выполняются все следующие операторы, включая и `операторDef`, на этом работа оператора выбора заканчивается. Если значения ни одной из констант не совпадают со значением выражения, программа выполняет `операторDef` по ветви `default`. Ветвь `default` может отсутствовать. При отсутствии совпадений со значениями `case` и отсутствии ветви `default` программа не выполняет никаких дальнейших действий.

Оператор `switch` отличается от оператора `if` тем, что он может выполнять проверку только равенства, в то время как оператор `if` может вычислять результат булевого выражения любого типа. То есть оператор `switch` ищет только соответствие между значением выражения и одной из констант `case`.

Как уже было сказано ранее, после выполнения оператора, отмеченного найденной константой, оператор `switch` продолжает выполнять все оставшиеся операторы в порядке их записи. Рассмотрим пример:

```
int a=2, b=2;  
switch (a){
```



```

    case 1: a++;
    case 2: a--;    //точка входа a=1
    case 3: a+=b;   //a=3
    case 4: a-=b;   //a=1
}

```

В данном фрагменте программы после вычисления оператора `a--` будут выполнены все оставшиеся операторы, после чего переменная `a` примет значение равное 1. Отметим, что если бы оператор `switch` в примере имел ветвь `default`, например такой:

```

. . .
default: a=0;}

```

то выполнялся бы оператор, находящийся в этой ветви, и значение переменной `a` обнулилось.

Если по алгоритму задачи необходимо выполнить список операторов только одной ветви `case`, то используют оператор прерывания `break`, который сразу же прекращает выполнение `switch` и передает управление оператору, следующему за оператором `switch`. Например:

```

int a=2, b=2;
switch (a){
    case 1: a++;
    case 2: a--;break;    //a=1
    case 3: a+=b;
    case 4: a-=b;    }

```

В примере выполниться оператор `a--`, отмеченный константой, равной 2, после чего выполниться оператор `break`, который завершит работу оператора `switch`.

Как видите, в операторе `switch` выполняется каждый оператор ветви `case`, пока не будет достигнут оператор `break` (или конец оператора `switch`).

Иногда бывает нужно выполнить один и тот же оператор в разных ветвях `case`. В этом случае ставят несколько меток `case` подряд. Например:

```

switch(dayOfWeek) {
case 1: case 2: case 3: case 4:case 5:
System.out.println("Рабочий день");, break;
case 6: case 7:
System.out.println("Выходной день"); break;
default:
System.out.println("Неверный ввод");
}

```

Оператор `switch` можно использовать в последовательности операторов внешнего оператора `switch`. Такой оператор называют вложенным оператором `switch`. Поскольку оператор `switch` определяет собственный блок, каких-либо конфликтов между константами `case` внутреннего и внешнего операторов `switch` не происходит, т.е. внутренний оператор `switch` и содержащий его внешний оператор `switch` могут иметь одинаковые константы `case`, как, например, в следующем фрагменте кода:

```
int a=2, d=0;
switch (a)
{
    case 1: d=a; break;
    case 2:
        switch (d)
        {
            case 0: a+=2; break;
            case 1: a-=2; break;
            case 2: a++; break;
        }
    case 3: d-=3; break;
}
```

Результат:

a=4 d=-3

Рассмотрим некоторые типичные ошибки применения оператора `switch`.

– Если в операторе `switch` используется выражение, следует обратить внимание на результирующий тип выражения (он должен быть целым, кроме `long`). Например, код:

```
double x = 1;
int y = 4;
switch(x*y){
case 2: ... break;
case 4: ... break;
default: ... }
```

вызовет ошибку компиляции. Исправить ошибку можно используя явное приведение типа: `switch((int)x*y)`.

– Если в операторе `switch` выражение имеет тип `byte` или `short`, то соответственно в операторах `case` должны быть указаны константы, которые могут быть сохранены в переменной данного типа. Например:

```
byte x = 5;
switch(x){
case 1: ... break;
```

```
case 132: ... break;
default: ... }
```

вызовет ошибку компиляции. Так как константа равная 132 превышает диапазон возможных значений типа byte.

В операторе switch не может быть двух case с одинаковыми константами. т.е. конструкция

```
switch(x) {
case 1: System.out.println("One"); break;
case 1: System.out.println("Two"); break;
case 3: System.out.println("Tree"); }
```

не допустима.

– В операторе switch может быть записана только одна ветвь default.

Задача 3.4. Для целого числа k от 1 до 99 напечатать фразу “Мне k лет”, учитывая при этом, что при некоторых значениях k слово “лет” надо заменить на слово “год” или “года”. Например, 11 лет, 22 года, 51 год.

Объяснение: если число заканчивается на цифру 1, то следует вывести слово «год», если на цифры 2, 3, 4 – «года». Исключение составляют только числа 11, 12, 13, 14, с ними употребляется слово «лет». Слово «лет» нужно выводить и со всеми оставшимися числами.

```
import java.util.Scanner;
class Ex_3_4 {
    public static void main (String[] args){
        Scanner s = new Scanner(System.in);
        System.out.println("Введите возраст от 1 до 99 ->");
        int k = s.nextInt();
        if (k>=11&&k<=14) System.out.printf("Мне %d лет",k);
        else switch (k%10)
        {
            case 1: System.out.printf("Мне %d год",k);
                    break;
            case 2: case 3: case 4:
                    System.out.printf("Мне %d года",k);
                    break;
            default: System.out.printf("Мне %d лет",k);
        }
    }
}
```

Задачу 3.4 можно было решить и не используя оператор switch, ограничившись только вложенными условными операторами if. Однако, как правило, оператор switch эффективнее набора вложенных операторов

`if` и в тех случаях, когда требуется осуществлять выбор в большой группе значений, оператор `switch` будет выполняться значительно быстрее последовательности операторов `if-else`. Это обусловлено тем, что компилятору известно, что все константы `case` имеют один и тот же тип, и их нужно просто проверять на равенство значению выражения `switch`.

Рассмотрим решения еще нескольких задач, в которых константные выражения в операторе `switch` имеют символьный и строковый типы.

Задача 3.5. Создать калькулятор с использованием оператора `switch`. Пользователь вводит два вещественных операнда и знак операции: `+` `-` `*` `/`, программа вычисляет результат соответствующего арифметического выражения и выводит его на экран.

Объяснение: значения операндов, введенных пользователем, будем сохранять в переменных `x` и `y`. Символ операции – в символьной переменной `operation`. Для сохранения результата арифметического выражения объявим вещественную переменную `result`.

В программе проводится проверка на равенство нулю делителя. В случае если при вычислении частного делитель `y` равен 0, выводится сообщение об ошибке и результату присваивается значение `Double.NaN` (не число).

Обратите внимание на то, что для считывания символов с клавиатуры используется метод `System.in.read()`. Это – одна из функций консольного ввода Java. Этот метод считывает символы из стандартного устройства ввода возвращаемые в виде целочисленных значений, поэтому тип возвращаемого значения был приведен к типу `char`. При использовании данного метода ввода программа должна содержать выражение `throws Exception`. Эта строка необходима для обработки ошибок ввода.

```
import java.util.Scanner;
class Ex_3_5 {
public static void main (String[] args) throws Exception{
    Scanner scan = new Scanner(System.in);
    System.out.println("Введите операнд1 ->");
    double x = scan.nextDouble();
    System.out.println("Введите операнд2 ->");
    double y = scan.nextDouble();
    System.out.println("Введите операцию (+,-,*,/) ->");
    char operation = (char) System.in.read();
    double result=0;
    switch (operation){
        case '+': result=x+y; break;
        case '-': result=x-y; break;
        case '*': result=x*y; break;
        case '/': if (y!=0) result=x/y; else
            {System.out.println("Ошибка! Деление на ноль");
```

```

        result=Double.NaN;} break;
    default:
        System.out.println("Неверный символ операции!");
    }
    System.out.printf("%5.2f%c%5.2f=%5.2f",x,operation,y,result);
}
}

```

Результат:

```

Введите операнд1 ->
5,2
Введите операнд2 ->
3,8
Введите операцию (+,-,*,/) ->
*
5,20* 3,80=19,76

```

Задача 3.6. Составьте программу, которая по введенному названию месяца выведет на экран название времени года, которому он принадлежит.

Объяснение: обратите внимание, что в программе для ввода и распознавания русского текста при создании объекта класса Scanner явно указана кодировка "UTF-8". Для работы со строковыми переменными используется тип String (в последующих разделах тип String будет рассмотрен более подробно).

```

import java.util.Scanner;
class Ex_3_6 {
    public static void main (String[] args){
        Scanner scan = new Scanner(System.in,"UTF-8");
        System.out.println("Введите месяц (например, март):");
        String month = scan.next();
        switch (month){
            case "декабрь":
            case "январь":
            case "февраль": System.out.println("Время года - зима");
                            break;

            case "март":
            case "апрель":
            case "май":      System.out.println("Время года - весна");
                            break;

            case "июнь":
            case "июль":
            case "август":   System.out.println("Время года - лето");
                            break;

            case "сентябрь":
            case "октябрь":
            case "ноябрь":   System.out.println("Время года - осень");
                            break;
        }
    }
}

```

```

    default: System.out.println("Неверный ввод!");
}
}}

```

Результат:

Введите месяц (например, март):

октябрь

Время года – осень

3.2 Операторы цикла

В программах часто приходится выполнять один и тот же оператор (или блок операторов) несколько раз. Для этого используют *операторы циклов*, которые реализуют алгоритмическую конструкцию повторения. Цикл состоит из заголовка и тела. В заголовке оператора цикла записывается *условие завершения* или *условие выполнения* цикла, а *тело цикла* представляет собой блок повторяющихся операторов. Каждое повторение операторов тела цикла называется *итерацией*.

В Java имеется три разновидности операторов цикла: оператор цикла с предусловием `while`, оператор цикла с пост условием `do-while` и оператор цикла с параметром `for`. В таблице 3.2 приведены их блок-схемы и краткое описание.

Таблица 3.2 – Разновидности операторов цикла в Java

Описание	Блок-схема
1. Цикл с предусловием (while) Предписывает выполнять тело цикла до тех пор, пока условие истинно	<pre> graph TD Start(()) --> Cond{условие} Cond -- Да --> Body[тело цикла] Body --> Cond Cond -- Нет --> Exit(()) </pre>
2. Цикл с постусловием (do – while) Предписывает выполнять тело цикла до тех пор, пока условие истинно.	<pre> graph TD Start(()) --> Body[тело цикла] Body --> Cond{условие} Cond -- Да --> Body Cond -- Нет --> Exit(()) </pre>
3. Цикл с параметром for Предписывает выполнять тело цикла для всех значений некоторой переменной (параметра цикла) в заданном диапазоне (от i1 до i2).	<pre> graph TD Init{{i=i1, i2}} --> Body[тело цикла] Body --> Init </pre>

3.2.1 Цикл с предусловием while

Общая форма оператора цикла while приведена ниже:

```
[инициализация;]
while (условие выполнения цикла)
{ //тело цикла
    [выражение итерации;]
}
```

Выполнение цикла while происходит следующим образом. Вначале вычисляется условие выполнения цикла, которое является булевым выражением, если его значение истинно, то выполняются операторы внутри цикла, т.е. тело цикла. Затем снова вычисляется условие и выполняются операторы, и так до тех пор, пока условие не станет ложным. Если условие изначально ложно, то тело цикла не будет выполнено ни разу. Например:

```
boolean b = false;
while(b) {
    System.out.println("Пустой вывод");}
```

В данном случае строка `System.out.println("Пустой вывод");` выполнена не будет.

Как можно видеть, в общей форме записи оператора цикла, кроме заголовка цикла `while(условие выполнения цикла)` имеется еще и необязательный раздел инициализация, а в теле цикла – выражение итерации. Дело в том, что любой цикл можно разделить на 4 части: инициализацию, тело, итерацию и условие выполнения. Перед входом в цикл `while` в первый раз обычно инициализируют одну или несколько переменных для того, чтобы условие выполнения цикла имело какое либо значение. Оператор или группа операторов, составляющих тело цикла, должны, как правило, с помощью выражения итерации изменять значения одной или нескольких переменных, входящих в условие выполнения, с тем, чтобы в конце концов оно стало равным `false` и цикл завершился. Если в цикл надо включить несколько операторов, то следует образовать блок операторов `{ }`.

Задача 3.7. Определите количество натуральных чисел, рассматривая их в порядке возрастания, сумма кубов которых не превышает N .

Объяснение: в программе необходимо последовательно суммировать кубы натуральных чисел 1, 2, 3, ..., и делать это до тех пор, пока сумма не достигнет целого значения N , введенного с клавиатуры. В результате нужно узнать, сколько чисел было пройдено.

Воспользуемся для решения задачи циклом с предусловием `while`. Введем обозначения: целая переменная `i` будет хранить текущее обрабатываемое натуральное число (1, 2, 3...). В теле цикла она будет увеличиваться на единицу, чтобы последовательно проходить числа, т.е. `i` – это переменная-счётчик.

Целая переменная `s` нужна для хранения суммы кубов чисел. В каждой итерации цикла необходимо увеличивать сумму `s` на куб текущего числа `i`.

Условием выполнения цикла будет условие самой задачи – достижение суммой `s` значения `N`.

```
import java.util.Scanner;
class Ex_3_7 {
    public static void main (String[] args){
        Scanner scan = new Scanner(System.in);
        System.out.println("Введите целое число N ->");
        int n = scan.nextInt();
        int i=0, s=0; //Инициализация
        while(s<=n) //Условие выполнения цикла
        {
            //Тело цикла
            i++; //Выражение итерации
            s+=Math.pow(i,3);
        }
        System.out.printf("Количество чисел = %d",i-1);
    }
}
```

Результат:

Введите целое число N ->

5000

Количество чисел = 11

Применение оператор цикла с предусловием особенно важно в тех случаях, когда необходимо организовать предварительную проверку, например, для того чтобы избежать деления на ноль. Цикл `while` может обеспечить безопасность выполнения кода в теле цикла.

Отметим также, что `while` позволяет организовать бесконечный цикл: `while (true) {тело цикла}`

Но помните, что из такого цикла следует обязательно предусмотреть какой-то выход, например, оператором `break` иначе программа заиклится.

Тело цикла может быть и пустым, например, в задаче 3.7. цикл `while` можно было записать следующим образом:

```
while((s+=Math.pow(++i,3))<=n) { }
```

и в этом случае результат выполнения программы остался бы тем же.

3.2.2 Цикл с постусловием do-while

В цикле с предусловием `while` первая проверка условия продолжения цикла происходит еще до первого выполнения его тела. Это означает, что при некоторых значениях параметров цикла его тело может не выполниться ни одного раза. Однако иногда желательно выполнить тело цикла хотя бы один раз, даже если в начальный момент условие выполнения ложно. В этом случае проверку условия прерывания цикла желательно выполнять в конце цикла, а не в его начале.

Цикл с постусловием `do-while` всегда выполняет тело цикла хотя бы один раз, поскольку условие его выполнения проверяется в конце цикла. Общая форма цикла `do-while` имеет следующий вид:

```
[ инициализация; ]
do { //тело;
    [выражение итерация;]
}
while (условие выполнения цикла);
```

При каждой итерации цикла `do-while` программа вначале выполняет тело цикла, а затем вычисляет условие выполнения цикла. Если условие истинно, то цикл повторяется. В противном случае выполнение цикла прерывается.

Задача 3.8. Написать программу расчета выражения: $f = a^3 + 3 \cdot \log(a)$, где $a > 0$. Организовать в программе проверку на “положительность” введенного значения a с помощью цикла `do-while`.

```
import java.util.Scanner;
class Ex_3_8 {
    public static void main (String[] args){
        Scanner scan = new Scanner(System.in);
        double a, f;
        do{
            System.out.println("Введите число a>0 ->");
            a=scan.nextDouble();
        }
        while(a<=0);
        f=Math.pow(a,3)+3*Math.log10(a);
        System.out.printf("f=%8.4f",f);
    }
}
```

Результат:

```
Введите число a>0 ->
0
```

```
Введите число a>0 ->  
-1,5  
Введите число a>0 ->  
-6,45  
Введите число a>0 ->  
-0,0025  
Введите число a>0 ->  
0,125  
f= -2,7073
```

Как вы видите, цикл с постусловием при любых обстоятельствах будет выполнен хотя бы один раз – в этом состоит его основное отличие от цикла с предусловием. Именно по этому признаку отдают предпочтение циклу с постусловием при решении определенного типа задач. В любом случае, и цикл с предусловием, и цикл с постусловием применяют тогда, когда количество повторений неизвестно до начала выполнения цикла – иначе используют цикл с параметром.

3.2.3 Цикл с параметром `for`

В случае, когда заранее известно количество повторений (итераций) цикла, зависящее от изменения значений какой-либо переменной в заданном диапазоне, удобно использовать цикл с параметром `for`.

Общая форма записи оператора `for` имеет вид:

```
for (инициализация; условие выполнения; выражение итерации)  
{  
    //тело;  
}
```

Если в цикле будет повторяться только один оператор, фигурные скобки можно опустить.

В круглых скобках цикла записаны: выражение инициализации, условие выполнения цикла и выражение итерации. Все эти разделы отделены друг от друга точкой с запятой. Рассмотрим их подробнее:

- выражение инициализации – выполняется до начала исполнения тела цикла и только один раз. Обычно в этом разделе записывается выражение, устанавливающее значение управляющей переменной – параметра цикла, которая действует в качестве счетчика, управляющего циклом.

- условие выполнения – булево выражение, до тех пор, пока оно истинно, тело цикла будет выполняться. Как правило, условие представлено логическим выражением, в котором сравнивается значение

управляющей переменной с целевым значением. Как и в случае с циклом `while` тело цикла может не исполниться ни разу, если данное условие принимает значение ложь до начала выполнения цикла.

– выражение итерации – выполняется сразу после исполнения тела цикла, и до того как проверено условие выполнения цикла. Обычно здесь используется выражение инкрементации или декрементации управляющей переменной, но может быть применено и любое другое выражение.

Цикл `for` работает следующим образом. При первом запуске цикла выполняется выражение инициализации, в котором обычно задается начальное значение параметра цикла. Затем вычисляется условие выполнения цикла. Если оно истинно, то выполняется тело цикла, а потом вычисляется выражение итерации. Далее снова проверяется условие выполнения цикла. Если оно истинно, то выполняется тело цикла и затем – выражение итерация и т. д. Как только условие станет ложным, выполнение цикла завершится.

Для примера рассмотрим цикл `for`, который находит сумму целых чисел в диапазоне от 1 до 100.

```
int sum=0;
for(int i=1; i<=100; i++)
    sum+=i;
```

В этом примере параметр цикла `i` инициализирован числом 1, а затем при каждой итерации сравнивается с числом 100. Пока переменная `i` меньше 100, цикл повторяется, текущее значение параметра `i` добавляется к переменной `sum`. Далее в разделе итерации `i` увеличивается на 1 и опять проверяется условие выполнения цикла `i<=100`. Данный процесс повторяется до тех пор, пока переменная `i` не станет больше 100. После этого цикл завершается, а управление передается оператору, следующему за ним. После выхода из цикла можно вывести значение переменной `sum`, которая будет равна сумме целых чисел в интервале от 1 до 100.

Обратите внимание – переменная `i` объявлена внутри раздела инициализации оператора `for`. Переменная, объявленная внутри оператора `for`, действует в пределах этого оператора, т.е. область видимости переменной будет ограничена телом цикла. За пределами тела цикла `i` является неизвестной переменной.

Рассмотрим некоторые ошибки, которые могут допускать начинающие программисты при работе с оператором цикла `for`.

Помните об объявленном типе переменной-параметра цикла и о его диапазоне значений. Например, следующий цикл будет выполняться бесконечно:

```
for(byte i=0;i<256;i++){
    ... }
```

Максимальное значение, которое может принять переменная типа `byte` равно 127, поэтому условие выполнения цикла всегда будет истинным. После превышения значения 127 переменной-параметру `i` будет присвоено значение -128, таким образом, значение 256 никогда не будет достигнуто.

Следует обратить внимание также на то, что раздел инициализации цикла `for` выполняется в любом случае, не зависимо от того будет выполнено тело цикла или нет. В качестве примера можно привести такую конструкцию:

```
int i=10;
for(i=0;i<0;i++){
    ...
}
System.out.println("i= " + i);
```

На экран будет выведено `i=0`, так как при запуске цикла сначала выполнилось выражение инициализации (параметр `i` стал равен 0), а затем уже проверка условия выполнения цикла.

Однако выражение итерации, напротив, выполняется только при выполнении тела цикла.

```
for(int i=0;i<1;i++){
    ...
}
System.out.println("i= " + i);
```

На экран будет выведено `i=1`.

Рассмотрим несколько простых задач, для решения которых удобно использовать оператор цикла `for`.

Задача 3.9. *Вычислить сумму всех четных натуральных чисел в диапазоне от 1 до 100 включительно.*

Объяснение: Первое четное число в заданном диапазоне – это 2, последнее – 100. Введем параметр цикла `i` с начальным значением равным 2. Каждое следующее четное число будем получать, увеличивая параметр `i` на 2. Цикл выполняется до тех пор, пока параметр `i` не превысит значения 100. Сумму четных элементов будем сохранять в переменную `sum`.

```
class Ex_3_9 {
    public static void main (String[] args){
        int sum=0;
        for (int i=0;i<=100;i+=2)
            sum+=i;
        System.out.printf("Сумма = %d",sum);
    }
}
```

Результат:

Сумма = 2550

Задача 3.10. Написать программу, которая возводит вещественное число *a* в натуральную степень *n*.

Объяснение: Возвести число в степень *n* можно путем умножения этого числа самого на себя *n* раз. Произведение будем сохранять в переменную *p*.

```
import java.util.Scanner;
class Ex_3_9 {
    public static void main (String[] args){
        Scanner scan = new Scanner(System.in);
        System.out.println("Введите a:");
        double a = scan.nextDouble();
        double p=1;
        System.out.println("Введите степь n:");
        int n = scan.nextInt();
        for (int i=1;i<=n;i++)
            p*=a;
        System.out.printf("%5.1f^%d=%5.3f",a,n,p);
    }
}
```

Результат:

```
Введите a:
5
Введите степь n:
2
5,0^2=25,000
```

3.2.4 Разновидности цикла for

В предыдущем разделе рассмотрена наиболее общая форма цикла `for`. Однако в языке Java допускаются некоторые его варианты, позволяющие во многих случаях увеличить мощность и гибкость программы.

Использование запятой. Допускается запись нескольких выражений в разделах инициализации и итерации внутри круглых скобок оператора цикла `for`. В этом случае их следует отделить друг от друга запятой. Выражения вычисляются слева направо, и в следующем уже можно использовать результат предыдущего выражения. Рассмотрим простые примеры, иллюстрирующие применение запятой в качестве разделителя нескольких выражений в разделах инициализации и итерации цикла.

Задача 3.11. Вывести на экран последовательность чисел:
0, -1, -4, -9, -16, -25

Объяснение: В следующем примере обе переменные (a и b) являются параметрами цикла for и обе инициализируются в этом цикле. Запятая разделяет два оператора инициализации. При каждой итерации значение переменной a увеличивается на 1, а значение b – уменьшается на 1. Для выполнения итерации как a, так и b должны быть инициализированы таким образом, чтобы выполнилось условие цикла при первой итерации.

```
class Ex_3_11 {
    public static void main (String[] args){
        for (int a=0, b=0;a-b<=10;a++,b--)
            System.out.printf("%d ",a*b);
    }
}
```

Результат: 0 -1 -4 -9 -16 -25

Задача 3.12. Вывести на экран 10 первых элементов последовательности $2 \cdot a_n - 2$, где $a_1 = 3$.

Объяснение: В следующем примере переменная i счетчик, который отсчитывает первых 10 элементов выводимой последовательности. Переменная a – выводимый на экран элемент.

```
class Ex_3_12 {
    public static void main (String[] args){
        for (int a=3, i=1;i<=10;a=2*a-2,i++)
            System.out.printf("%d ",a);
    }
}
```

Результат: 3 4 6 10 18 34 66 130 258 514

Следует отметить, что при использовании в цикле for нескольких выражений в разделе инициализации, невозможно объявление нескольких переменных разных типов. Так при попытке использования следующих выражений будет получена ошибка компиляции.

```
for(int i=0, long j=0;i<10;i++,j+=50) // неверно
//можно заменить на

int i; long j;
for(i=0, j=0;i<10;i++,j+=50)
```

Однако возможно объявить и инициализировать переменные одного типа, например, в следующем примере объявлены сразу две переменные типа `int` – `i` и `j`:

```
for(int i=0, j=0; i<10; i++, j+=10)
...
```

Но если переменная уже объявлена до цикла, то повторное объявление ее в разделе инициализации вызовет ошибку:

```
int j;
for(int i=0, j=0; i<10; i++, j+=10) //ошибка!
...
```

Цикл `for` без тела цикла. Приемлемой является запись оператора `for` с отсутствующим телом цикла. В следующем примере находится сумма первых десяти натуральных чисел.

```
int sum=0;
for (int i=0; i<10; i++, sum+=i);
System.out.printf("%d ", sum);
```

Как вы видите, тело цикла отсутствует. После закрывающей круглой скобки заголовка цикла стоит точка с запятой. Будьте внимательны, выражения в разделе итерации выполняются слева направо. Это значит, что параметр `i` сначала увеличится на 1, а потом его новое значение добавится к сумме `sum`. С учетом этого, условием выполнения цикла должно быть булево выражение `i<10`. Если выражения итерации поменять местами, то следует изменить и условие выполнения цикла:

```
for (int i=0; i<=10; sum+=i, i++);
```

Бесконечный цикл. Любой раздел оператора `for` может отсутствовать, при этом точки с запятой сохраняются. Например, можно задать бесконечный цикл:

```
for (;;) { ... }
```

Эта конструкция аналогична рассмотренной ранее конструкции `while(true){...}`. В этом случае в теле цикла следует предусмотреть какой-нибудь выход.

Вложенные циклы. В языке Java допускает использование вложенных циклов. Другими словами, один цикл может выполняться внутри другого. Например, в следующих программах использованы вложенные циклы `for`.

Задача 3.13. Найдите все целые положительные трехзначные числа, у которых сумма цифр равна их произведению.

Объяснение: трехзначные числа состоят из трех цифр, обозначающих сотни, десятки и единицы. Причем цифры десятков и единиц изменяются в диапазоне $[0 \div 9]$, а цифра сотен – в диапазоне $[1 \div 9]$. В программе используются три вложенных цикла `for`, которые задают диапазоны изменений цифр сотен `k1`, десятков `k2` и единиц `k3` трехзначного числа `s`

шагом 1. Простым перебором находятся трехзначные числа, цифры которых удовлетворяют условию задачи.

```
public class Ex_3_13 {  
    public static void main(String[] args) {  
        int k1,k2,k3;  
        for (k1 = 1; k1 <= 9; k1++)  
            for (k2 = 0; k2 <= 9; k2++)  
                for (k3 = 0; k3 <= 9; k3++)  
                    if (k1+k2+k3==k1*k2*k3)  
                        System.out.println((k1*100+k2*10+k3));  
    }  
}
```

Результат:

```
123  
132  
213  
231  
312  
321
```

3.3 Операторы перехода

В языке Java определены три оператора перехода: `break`, `continue` и `return`. Они передают управление другой части программы. Рассмотрим подробнее операторы `break` и `continue`. Оператор `return` будет рассмотрен в последующих главах.

3.3.1 Оператор `break`

Как уже было сказано ранее, оператор `break` используется тогда, когда нужно завершает выполнение оператора `switch`. Кроме того оператор `break` позволяет вызвать немедленное завершение цикла, пропуская условное выражение и последовательность операторов в теле цикла. Как только программа доходит до оператора `break` внутри тела цикла, она прекращает выполнение цикла и управление передается оператору, следующему за циклом.

Например, запишем цикл, который найдет наименьшее двузначное число, куб которого больше числа `n`.

```
int i;  
for(i=10; i<=99; i++)  
    if (Math.pow(i,3)>n) break;
```

Как видите, хотя цикл `for` должен был бы выполняться для значений параметра `i` от 10 до 99, оператор `break` приводит к более

раннему выходу из него, когда значение параметра в кубе становится больше значения n .

Часто оператор `break` используют для выхода из бесконечного цикла. Например, в следующем фрагменте программы запрашивается ввод пароля до тех пор, пока пользователь не введет число 12345.

```
import java.util.Scanner;
public class Example {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int password;
        while(true)
        { System.out.println("Введите пароль:");
          password = scan.nextInt();
          if (password==12345) break;
        }
        System.out.println("Пароль успешно введен!");
    }
}
```

Если оператор `break` используется внутри набора вложенных циклов оператор, то он осуществляет выход только из самого внутреннего цикла. На внешний цикл он не оказывает никакого влияния.

В следующем примере внешний цикл `for` задает количество строк при выводе символа "*", а внутренний цикл `for` выводит сам символ в текущую строку. Количество выводимых в каждую строку символов "*" ограничивается условием `i==j`, после чего внутренний цикл прерывается оператором `break`. Таким образом, количество символов выведенных в строке соответствует номеру данной строки.

```
public class Example {
    public static void main(String[] args) {
        for(int i=1; i<=10; i++){
            for(int j=1; j<=10; j++)
            { System.out.print("*");
              if(i==j) break;}
            System.out.println(); }
    }
}
```

Результат:

```
*
**
***
****
*****
*****
*****
*****
```

```
*****
*****
*****
```

3.3.2 Оператор continue

В некоторых ситуациях возникает потребность досрочно перейти к выполнению следующей итерации, проигнорировав часть операторов тела цикла, еще не выполненных в текущей итерации. Для этой цели в Java предусмотрен оператор `continue`. Оператор `continue` используется только в операторах цикла и осуществляет немедленный переход к следующей итерации цикла. Например, в следующем фрагменте кода оператор `continue` позволяет обойти деление на нуль:

```
for (int i=0;i<n;i++){
    if (i==j) continue;
    sum+=1.0/(i-j);
}
```

В циклах `while` и `do-while` оператор `continue` сразу передает управление на условие выполнения цикла. В цикле `for` управление передается вначале итерационной части цикла `for`, а потом условию выполнения.

Задача 3.14. Организовать ввод 5-ти вещественных чисел. Найти сумму только положительных из введенных чисел.

```
import java.util.Scanner;
public class Ex_3_14 {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        double sum=0;
        double x;
        for(int i=1; i<=10; i++){
            System.out.printf("Введите %d-е число:%n", i);
            x=scan.nextDouble();
            if(x<=0) continue;
            sum+=x;}
        System.out.printf("sum=%5.2f", sum);
    }
}
```

Результат:

```
Введите 1-е число:
5
Введите 2-е число:
9
Введите 3-е число:
```

```
-7
Введите 4-е число:
-5
Введите 5-е число:
-4
sum=14,00
```

3.3.3 Организация безусловного перехода в Java

Часто при использовании вложенных циклов может возникнуть ситуация когда из вложенного цикла нужно прервать внешний цикл. Это нельзя сделать простым оператором `break` или `continue`, однако в Java возможно именовать блок кода и явно указать этим, к какому из блоков относится прерывание. Метку можно использовать как с самостоятельным блоком кода, так и с блоком другого оператора, например, `do`, `while` и `for`. В случае если при выполнении операторов `break` или `continue` управление следует передать помеченному меткой блоку кода, используют следующие формы данных операторов:

```
break метка;
continue метка;
```

Метка – это любой допустимый идентификатор Java, за которым следует двоеточие. Как только блок или цикл помечен, его метку можно использовать в качестве цели оператора `break` или `continue`.

Рассмотрим примеры использования меток для организации безусловного перехода.

```
for(int i=0;i<n;i++){
for(int j=0;j<m; j++){
...
if(i==j) break;
...
}}
```

В данном случае, при выполнении условия будет прервано выполнение цикла по параметру `j`, цикл по параметру `i` продолжится со следующего значения. Тот же результат может быть получен и оператором `continue` с меткой.

```
M1: for(int i=0;i<n;i++){
    for(int j=0;j<m; j++){
        ...
        if(i==j) continue M1;
        ...
    }
```

```
}}
```

Для того, что бы прервать выполнение обоих циклов, следует использовать оператор `break` с меткой:

```
M1: for(int i=0;i<n;i++){
    for(int j=0;j<m; j++){
        ...
        if(i==j) break M1;
        ...
    }
}
```

Между операторами `break` и `continue` есть одно существенное отличие. Оператор `break` может быть использован с любым именованным блоком. Оператор `continue` (как и отмечалось ранее) может быть использован только в теле цикла. Например, такая конструкция будет вполне приемлемой:

```
M1:{
    ...
    if(a>0) break M1;
    ...
}
```

в то время как оператор `continue` здесь применять нельзя. В данном случае если условие оператора `if` будет истинным, выполнение блока с меткой `M1` будет прервано, а управление будет передано на оператор, следующий непосредственно за закрывающей фигурной скобкой.

Обратите внимание на то, что оператор `break` с меткой можно применять для выхода из набора вложенных блоков. Однако его нельзя использовать для передачи управления внешнему блоку кода, который не содержит данный оператор `break`. Например, следующий фрагмент кода ошибочный:

```
M1: for(int i=0;i<n;i++){
    ...
}
    for(int j=0;j<m; j++){
        ...
        if(i==j) break M1;
        ...
    }
```

Блок, помеченный меткой `M1`, не содержит оператор `break`, поэтому передача управления этому внешнему блоку невозможна.

3.4 Некоторые циклические алгоритмы и программы

Рассмотрим использование алгоритмов циклической структуры на конкретных примерах. Некоторые циклические алгоритмы уже были представлены ранее в качестве примеров, демонстрирующих выполнение различных операторов цикла. В данном разделе, кроме новых задач, рассмотрим важное применение циклических структур – генерирование рекуррентных последовательностей.

3.4.1 Алгоритм определения простого числа

Из курса математики известно, что натуральное число n называется *простым*, если оно делится нацело без остатка только на единицу и n . Например, число 13 – простое, так как делится только на 1 и 13, а число 12 не является простым, так как делится на 1, 2, 3, 4, 6 и 12.

Алгоритм проверки натурального числа на простоту является циклическим, так как предполагает повторение следующих действий: проверка делимости числа n на делитель k и выбор следующего делителя путем увеличения k на единицу. Начальным значением параметра цикла k является 2, так как на единицу делятся все числа. Условием продолжения цикла является условие $k < n$. Следует так же отметить, что кажущийся на первый взгляд очевидным перебор значений параметра цикла от 2 до $n-1$ является не целесообразным, так как значение наибольшего делителя составного натурального числа n не может превышать величины $\sqrt{n}$. Следовательно, в цикле достаточно будет в качестве делителей перебрать целые числа в диапазоне от 2 до $\lfloor \sqrt{n} \rfloor$ ²³.

В каждой итерации цикла следует находить остаток от деления n на k . Равенство нулю остатка от деления сигнализирует о том, что найден делитель и можно досрочно закончить выполнение цикла с помощью оператор перехода `break`.

Для того, чтобы после выхода из цикла понять, был ли цикл прерван принудительно по причине нахождения делителя или завершился самостоятельно, после того как условие его выполнения стало ложным, удобно использовать специальную логическую переменную-флаг. В начале выполнения цикла ей присваивается значение `true`, а после нахождения делителя – `false`. Таким образом, после завершения цикла считаем, что число простое, если значение флага истинно, в противном случае – число составное. Блок-схема алгоритма определения простого числа приведена на рис. 3.4.

²³ Символом $\lfloor x \rfloor$ в математике обозначается целая часть числа x .

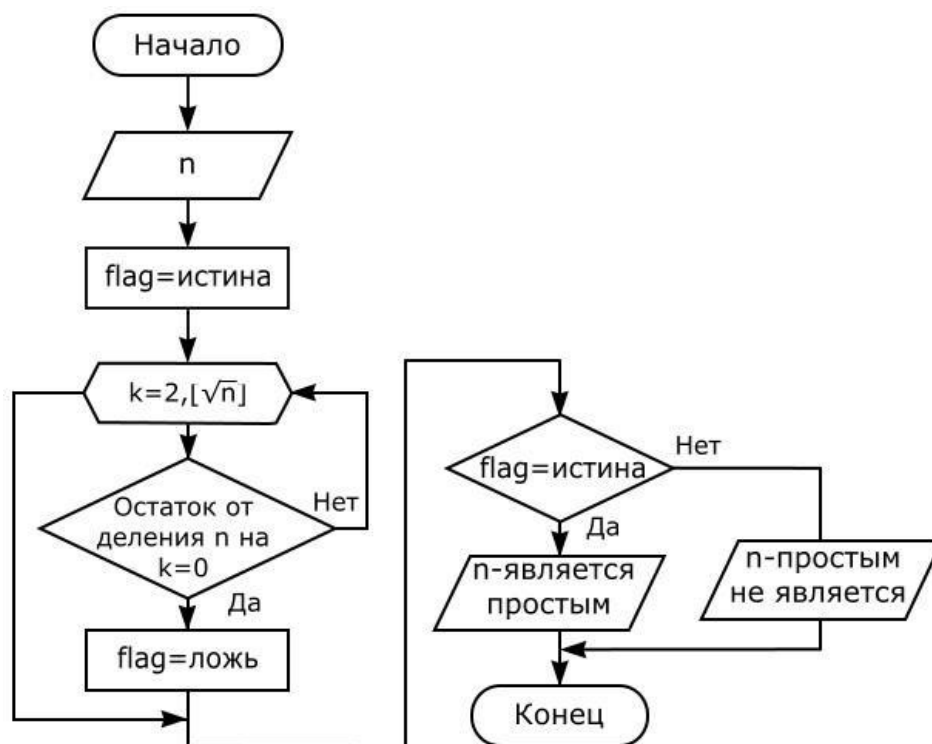


Рис.3.4 – Блок схема алгоритма определения простого числа

Задача 3.15. Дано натуральное число n . Определите, является ли оно простым.

```

import java.util.Scanner;
public class Ex_3_15 {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        boolean flag=true;
        System.out.println("Введите натуральное число:");
        int n=scan.nextInt();
        for(int k=2; k<Math.sqrt(n); k++)
            if(n%k==0) {flag=false; break;}
        if(flag)
            System.out.printf("число %d - простое", n);
        else
            System.out.printf("число %d - составное", n);
    }
}
  
```

Результат

Введите натуральное число:

13

число 13 - простое

3.4.2 Алгоритм нахождения наибольшего общего делителя

Наибольший общий делитель (НОД) двух натуральных чисел a и b – это самое большое число, на которое оба данных числа a и b делятся без остатка. Чтобы найти НОД двух чисел нужно: разложить эти числа на простые множители, а потом найти произведение одинаковых простых множителей. Например, найдем НОД для чисел $a=28$ и $b=64$. Разложим их на простые множители:

$$28 = \underline{2} \cdot \underline{2} \cdot 7$$

$$64 = \underline{2} \cdot \underline{2} \cdot \underline{2} \cdot \underline{2} \cdot \underline{2} \cdot \underline{2}$$

и найдем произведение одинаковых множителей: $\text{НОД}(28;64) = 2 \cdot 2 = 4$.

В программировании для нахождения НОД применяют *алгоритм Евклида*, который выполняется в такой последовательности:

- 1) из большего числа вычитаем меньшее;
- 2) если получается 0, то числа равны друг другу и являются НОД, работа алгоритма завершается;
- 3) если результат вычитания не равен 0, то большее число заменяем на результат вычитания;
- 4) переходим к пункту 1.

Таким образом, в соответствии с алгоритмом Евклида, следует каждый раз уменьшать большее из чисел на величину меньшего до тех пор, пока оба значения не станут равными. Рассмотрим таблицу выполнения алгоритма Евклида для нахождения НОД чисел $a=28$ и $b=64$ (табл.3.3).

Таблица 3.3 – Поиск НОД чисел $a=28$ и $b=64$ с помощью алгоритма Евклида

Итерация	a	b	НОД(a,b)
Исходные данные	28	64	
$64-28=36$	28	36	
$36-28=8$	28	8	
$28-8=20$	20	8	
$20-8=12$	12	8	
$12-8=4$	4	8	
$8-4=4$	4	4	4

В приведенном выше алгоритме НОД находится вычитанием чисел. Существует более эффективная разновидность этого алгоритма, где НОД находится делением:

- 1) большее число делим на меньшее;

- 2) если делится без остатка, то меньшее число и есть НОД и работа алгоритма завершается;
- 3) если есть остаток, то большее число заменяем на остаток от деления;
- 4) переходим к пункту 1.

В табл.3.4 представлен поиск НОД чисел $a=28$ и $b=64$ с помощью деления. Блок-схема алгоритма изображена на рис.3.5.

Таблица 3.4 – Поиск НОД чисел $a=28$ и $b=64$ с помощью деления

Итерация	a	b	НОД(a,b)
Исходные данные	28	64	
$64\%28=8$	28	8	
$28\%8=4$	4	8	
$8\%4=0$	4	8	4

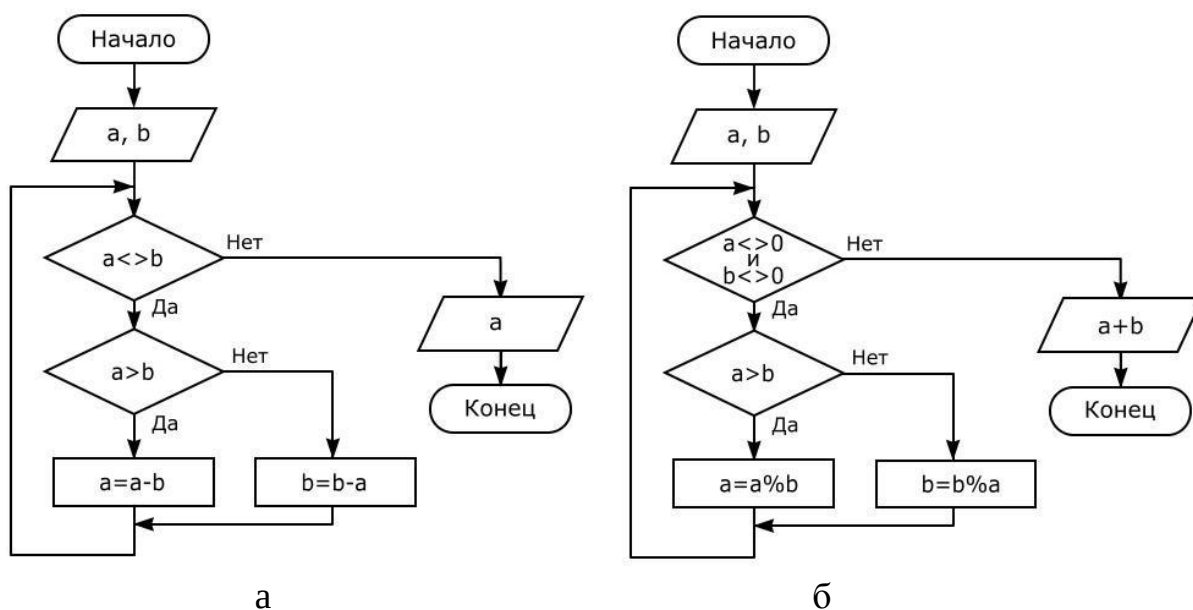


Рис. 3.5 – Блок-схема алгоритма поиска НОД двух чисел: вычитанием (а); делением (б)

Задача 3.16. Найдите НОД двух натуральных чисел a и b .

Алгоритм Евклида, основанный на вычитании чисел:

```

import java.util.Scanner;
public class Ex_3_16 {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
  
```



```

        System.out.println("Введите число a:");
        int a=scan.nextInt();
        System.out.println("Введите число b:");
        int b=scan.nextInt();
        while (a!=b) {
            if (a>b) a-=b;
            else b-=a; }
        System.out.println("НОД= "+a);
    }}

```

Алгоритм Евклида, основанный на делении чисел:

```

import java.util.Scanner;
public class Ex_3_16 {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Введите число a:");
        int a=scan.nextInt();
        System.out.println("Введите число b:");
        int b=scan.nextInt();
        while (a!=0 && b!= 0)
            { if (a >= b) a%=b;
              else b%=a;}
        System.out.println("НОД= "+(a+b));
    }}

```

3.4.3 Рекуррентные последовательности

В математике очень часто встречаются последовательности чисел, в которых каждый следующий член выражается через предыдущие. В арифметической прогрессии, например, каждый следующий член равен предыдущему, увеличенному на разность прогрессии: $a_n = a_{n-1} + d$.

Формулы, выражающие очередной член последовательности через один или несколько предыдущих членов, называют *рекуррентными соотношениями*. Последовательность, члены которой удовлетворяют некоторому рекуррентному соотношению, называется *рекуррентной*.

Рассмотрим наиболее известные рекуррентные последовательности и соотношения.

Факториал числа n равен произведению всех натуральных чисел до n включительно: $n! = 1 \cdot 2 \cdot \dots \cdot n = \prod_{i=1}^n i$. По определению полагают $0! = 1$. На основе этого можно записать: $1! = 0! \cdot 1$; $2! = 1! \cdot 2$; $3! = 2! \cdot 3$; К ; $n! = (n-1)! \cdot n$. Теперь можно определить рекуррентное соотношение для факториала $n!$:

$$\begin{aligned}
 0! &= 1, \\
 n! &= n \cdot (n - 1)! \quad \text{для } n > 0.
 \end{aligned}
 \tag{3.1}$$

Факториал определен только для целых неотрицательных чисел.

В комбинаторике факториал натурального числа n интерпретируется как количество перестановок множества из n элементов. Например, для множества $\{A, B, C\}$ существует $3! = 6$ перестановок: ABC, ACB, BAC, BCA, CAB, CBA.

Ниже приведена блок-схема вычисления факториала натурального числа n .

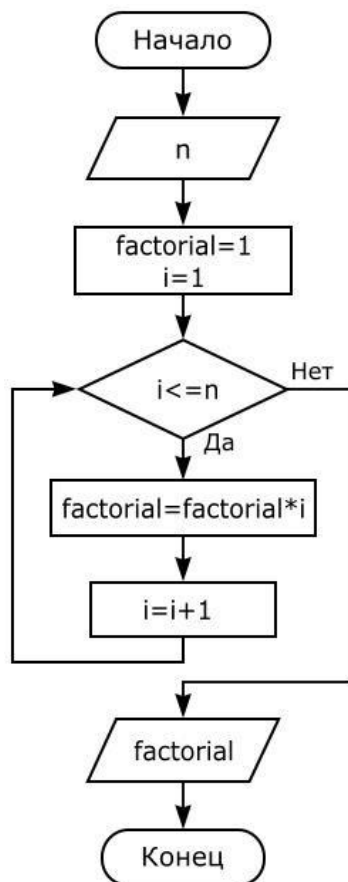


Рис. 3.6 – Блок-схема алгоритма вычисления факториала

Задача 3.17. Определите факториал целого неотрицательного числа n .

Объяснение: Программа реализует рекуррентное соотношение (3.1) для вычисления факториала. В теле цикла `while` происходит присваивание переменной `factorial` произведения ее предыдущего значения и текущего значения параметра-счетчика цикла `i`. Так как функция факториала очень быстро возрастает, в программе предусмотрено

ограничение на ввод ($n < 20$) для того, чтобы не превысить наибольшее допустимое значение типа long.

```
import java.util.Scanner;
public class Ex_3_17 {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int n, i=1;
        long factorial=1;
        do{
            System.out.println("Введите число n (n<17):");
            n=scan.nextInt();
        }while (n>=20);
        while(i<=n)
            {factorial*=i; i++;}
        System.out.printf("%d!=%d",n,factorial);
    }
}
```

Результат:

Введите число n (n<20):

60

Введите число n (n<20):

9

9!=362880

Числами Фибоначчи называется последовательность f_i вида: 1, 1, 2, 3, 5, 8, 13, 21, ..., которую описал итальянский математик Леонардо Фибоначчи, решая задачу о размножении кроликов. Нетрудно видеть, что $f_1 = 1$, $f_2 = 1$, а каждое следующее число, начиная с третьего, равно сумме двух предыдущих. Таким образом, все члены последовательности чисел Фибоначчи задаются рекуррентным соотношением:

$$\begin{aligned} f_1 &= 1, f_2 = 1, \\ f_i &= f_{i-2} + f_{i-1}. \end{aligned} \quad (3.2)$$

Числа Фибоначчи имеют ряд 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.

Рассмотрим алгоритм вычисления n -го числа Фибоначчи с помощью цикла. Для того чтобы сгенерировать последовательность из n чисел Фибоначчи, необходимо реализовать рекуррентное соотношение (3.2). Так как текущее значение числа, начиная с третьего, равно сумме двух предыдущих, то введем следующие переменные: $f1$ и $f2$ для хранения значений двух предыдущих значений ряда, начальные их значения равны 1 и f – текущее значение последовательности. В теле

цикла после вычисления текущего числа Фибоначчи: $f=f1+f2$, выполняется переприсваивание значений переменных $f1$ и $f2$ для того, чтобы они содержали новую пару чисел Фибоначчи: $f1=f2$ и $f2=f$. Кратко алгоритм можно представить в виде следующих шагов:

- 1) Ввести два начальных значения ряда ($f1$ и $f2$).
- 2) Ввести номер определяемого элемента (n).
- 3) Выполнять нижеследующие действия количество раз, равное по величине номеру определяемого элемента, уменьшенному на две единицы (т.к. первое и второе значение ряда уже известны):
 - сложить $f1$ и $f2$, присвоив результат третьей переменной (f).
 - поменять начальные значения: $f1 = f2$, а $f2 = f$.

Блок-схема алгоритма вычисления чисел Фибоначчи представлена на рис. 3.7.

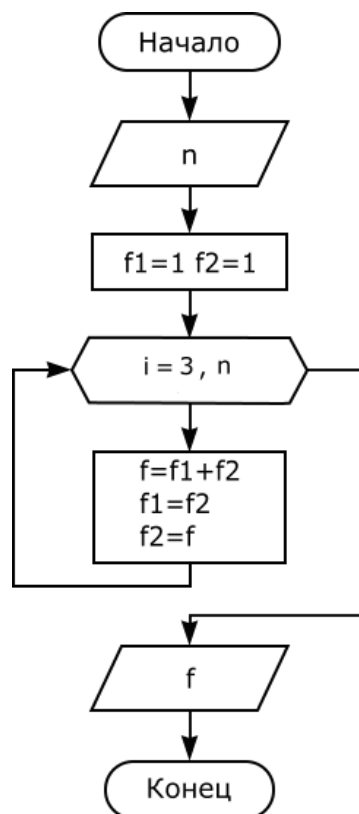


Рис. 3.7 – Блок-схема алгоритма вычисления n -го числа Фибоначчи

Задача 3.18. Определите n -ое число Фибоначчи.

```
import java.util.Scanner;
public class Ex_3_18 {
    public static void main(String[] args) {
```

```

Scanner scan = new Scanner(System.in);

long f1=1, f2=1, f=0;
System.out.println("Введите число n:");
int n=scan.nextInt();
for(int i=3; i<=n; i++)
    {f=f1+f2;
      f1=f2;
      f2=f;}
System.out.printf("%d-е число Фибоначчи = %d",n,f);
}}

```

Рассмотрим еще один пример рекуррентной последовательности.

Задача 3.19. Для заданного натурального числа n и вещественного x ($x > 0$) вычислить значение выражения: $\sqrt{x + \sqrt{x + K + \sqrt{x}}}$.

Объяснения: решение можно свести к вычислению n -го элемента последовательности следующего вида:

$$a_1 = \sqrt{x}, \quad a_2 = \sqrt{x + \sqrt{x}}, \quad a_3 = \sqrt{x + \sqrt{x + \sqrt{x}}}, \dots$$

Отсюда можно получить рекуррентное соотношение:

$$a_2 = \sqrt{x + a_1}, \quad a_3 = \sqrt{x + a_2}, \quad a_i = \sqrt{x + a_{i-1}}.$$

Запишем реализацию полученных рекуррентных соотношений на языке Java

```

import java.util.Scanner;
public class Ex_3_19 {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Введите x:");
        double x=scan.nextDouble();
        System.out.println("Введите n:");
        int n=scan.nextInt();
        double a=Math.sqrt(x);
        for(int i=2; i<=n; i++)
            a=Math.sqrt(x+a);
        System.out.printf("Значение выражения (для n=%d и x=%5.3f) равно %5.3f",n,x,a);
    }}

```

Результат:

Введите x:

0,52

Введите n:

6

Значение выражения (для n=6 и x=0,520) равно 1,373

3.4.4 Схема Горнера

Рассмотрим еще один важный пример – функция на последовательности. Пусть дана последовательность коэффициентов многочлена $p(x)$ по убыванию степеней вида:

$$p(x) = a_0 x^n + a_1 x^{n-1} + K + a_n$$

Нужно вычислить значение многочлена в точке $x = t$. Алгоритм, основанный на просмотре последовательности коэффициентов в направлении от старшего к младшему, называется *схемой Горнера*. Проиллюстрируем его идею на примере многочлена третьей степени:

$$p(x) = a_0 x^3 + a_1 x^2 + a_2 x + a_3$$

Его можно представить в виде:

$$p(x) = ((a_0 x + a_1)x + a_2)x + a_3$$

Для вычисления значения многочлена достаточно трех умножений и трех сложений. В общем случае, многочлен представляется в следующем виде:

$$p(x) = (K((a_0 x + a_1)x + a_2)x + K + a_{n-1})x + a_n \quad (3.3)$$

Обозначим через $p_k(x)$ многочлен k -ой степени, вычисленный по коэффициентам $a_0, a_1, \dots, a_k$:

$$p_k(x) = a_0 x^k + a_1 x^{k-1} + K + a_k$$

Тогда можно записать рекуррентное соотношение:

$$p_{k+1}(x) = p_k(x) \cdot x + a_{k+1} \quad (3.4)$$

Таким образом, при считывании нового коэффициента многочлена надо старое значение многочлена умножить на значение x , а затем прибавить к нему новый коэффициент.

Задача 3.20. Вычислите по схеме Горнера значения многочлена вида: $y = x^{10} + 2x^9 + 3x^8 + K + 10x + 11$.

Объяснение: используя соотношение (3.3) для заданного в задаче многочлена можно определить $p_1(x) = x + 2$, тогда в соответствии с (3.4)

для нахождения значения многочлена достаточно в каждой итерации цикла умножать $p_1(x)$ на x и прибавлять очередной коэффициент.

```
import java.util.Scanner;
public class Ex_3_20 {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Введите x:");
        double x=scan.nextDouble();
        double y=x+2;
        for(int i=3; i<=11; i++)
            y=y*x+i;
        System.out.printf("Значение многочлена равно %5.3f", y);
    }
}
```

Результат:

Введите x:

0,32

Значение многочлена равно 15,484

3.4.5 Степенные ряды

Рассмотрим итерационную задачу вычисления приближенных значений математических функций путем нахождения частичной суммы соответствующего ряда Тейлора. При решении такой задачи вычисления проводятся до тех пор, пока не будет достигнута заданная точность, при этом заранее не известно количество суммируемых членов ряда.

Рассмотрим пример приближенного вычисления значения функции $y = e^x$, представленной в виде степенного ряда Тейлора:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!} + \dots, \quad n = 1, 2, \dots$$

Для вычисления значения функции e^x с точностью $\varepsilon > 0$ необходимо просуммировать все члены ряда, которые по модулю превышают заданную точность ε . Следовательно, для рассматриваемого ряда необходимо организовать циклический процесс для вычисления суммы и закончить его, когда выполнится условие:

$$\left| \frac{x^n}{n!} \right| < \varepsilon.$$

Для вычисления члена ряда можно использовать рекуррентное соотношение, что является намного более эффективным, чем вычислять

факториал и выполнять возведение в степень. Можно заметить, что соседние члены ряда связаны между собой следующим соотношением:

$$a_i = a_{i-1} \cdot \frac{x}{i}, \text{ где } i=1, 2, K, \text{ причем } a_0 = 1. \quad (3.5)$$

Используя это соотношение можно последовательно вычислить все члены ряда, не используя в явном виде операций вычисления факториала и возведения в степень. Блок – схема алгоритма представлена на рис.3.8.

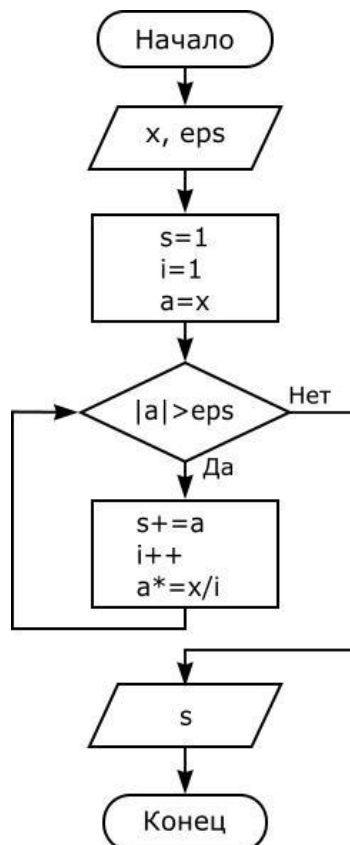


Рис.3.8 – Блок-схема алгоритма вычисления функции $y = e^x$

Задача 3.21. Вычислить приближенное значение функции $y = e^x$, используя ее разложения в ряд Тейлора, с заданной точность $eps > 0$.

Объяснение: на экран выводится сумма ряда, рассчитанная в соответствии с рекуррентным соотношением (3.5) и значение функции, полученное путем вызова метода `exp()` класса `Math`.

```

import java.util.Scanner;
public class Ex_3_21 {
    public static void main(String[] args) {

```



```

Scanner scan = new Scanner(System.in);
System.out.println("Введите x:");
double x=scan.nextDouble();
System.out.println("Введите точность:");
double eps=scan.nextDouble();
double s=1, i=1, a=x;
while(Math.abs(a)>eps){
    s+=a; i++; a*=x/i;}
System.out.printf("s=%8.6f      exp(x)=%8.6f
error=%5.2g",s,Math.exp(x), Math.abs(s-Math.exp(x)));
}

```

Результат:

Введите x:

2,25

Введите точность:

0,00001

s=9,487729 exp(x)=9,487736 error=7,2e-06

Упражнения к главе 3

1. Определите значения переменных x , y и z после выполнения фрагмента программы:

```

a) int x=2, y=5, z=1;
   while (x+y>0) {
       x+=2; y-=x; z++;}

```

Построим трассировочную таблицу:

Итерация	x	y	z	Условие выполнения цикла
Начальное значение	2	5	1	True
1	4	1	2	True
2	6	-5	3	True
3	8	-13	4	False

Ответ: $x=8$; $y=-13$; $z=4$.

```

b) int x=12, y=3, z=1;
   while (x/y>0) {
       x--; y+=2; z++;}

```

```

c) double x=0.5, y=3, z=1.;
   do {
       x*=3; y-=0.2; z++;}

```

```
while (x<=y) ;
```

```
d) double x=0.5, y=3.5, z=0;
   do {
       x*=2; y-=4; z++; }
   while (x+y>0);
```

2. Известно, что из четырех чисел a_1, a_2, a_3 и a_4 одно отлично от трех других, равных между собой; присвоить номер этого числа переменной n .

3. Выведите на экран числа в следующем виде:

```
7 6 5 4 3 2
6 5 4 3 2
5 4 3 2
4 3 2
3 2
2
```

4. Запишите рекуррентные соотношения необходимые для вычислений функций:

a) $y = 1 - x^3 + x^6 - x^9 + K$

b) $y = 1 - x^2 - x^4 - x^6 - K$

c) $y = \frac{1}{x} + \frac{2}{x^2} + \frac{3}{x^3} + \frac{4}{x^4} + K$

d) $y = \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + K$

5. Составив соответствующее рекуррентное соотношение, вычислите значения выражений:

a) $1 + \frac{1}{2!} + \frac{1}{3!} + K + \frac{1}{n!}$

b) $\sqrt[4]{2 + \sqrt[4]{2 + K + \sqrt[4]{2 + \sqrt[4]{2}}}}$

c) $\sin x + \sin x^2 + K + \sin x^n$

d) $\sin(x) + \sin(\sin(x)) + K + \sin(\sin(\sin(x)))$

6. Напишите программу, которая запрашивает у пользователя числа до тех пор, пока каждое следующее число больше предыдущего. В конце программа сообщает, сколько чисел было введено.

7. Подсчитайте количество цифр в десятичной записи целого неотрицательного числа k .

8. Вычислите $n!! = \begin{cases} 1 \cdot 3 \cdot 5 \cdot K \cdot n, & \text{if } n \text{ is odd} \\ 2 \cdot 4 \cdot 6 \cdot K \cdot n, & \text{if } n \text{ is even} \end{cases} \quad n > 0.$

9. Определите количество простых чисел в интервале от n до m , где n и m – натуральные числа.
10. Найдите первое число Фибоначчи больше числа m .
11. Вычислите по схеме Горнера значение многочлена вида:

$$y = 11x^{10} + 10x^9 + 9x^8 + K + 2x + 1$$

12. В 1674 году Г.Лейбниц показал, что число $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + K$

Найдите приближенное значение числа π , просуммировав 100 членов этого ряда.

13. Известно, что последовательность сумм вида:

$$s_n = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + K + (-1)^n \frac{x^{2n}}{(2n)!}$$

сходится к функции $\cos(x)$. Напишите программу нахождения суммы ряда с точность $\varepsilon = 10^{-6}$.

14. Известно, что последовательность сумм вида:

$$s_n = x - \frac{x^3}{3!} + \frac{x^5}{5!} - K + (-1)^{n+1} \frac{x^{2n-1}}{(2n-1)!}$$

сходится к функции $\sin(x)$. Исследуйте зависимость от x точности приближения синуса десятым членом последовательности. Для этого рассчитайте величину $|\sin(x) - s_{10}(x)|$ в 20 точках в диапазоне от 0 до 2π .

4 МАССИВЫ

Рассмотрим один из важных ссылочных типов данных – *массивы*.

В отличие от обычных переменных, которые хранят только одно значение, массивы используются для хранения целого набора значений. Их удобно использовать там, где нужно организовать обработку относительно большого количества переменных одного типа. Представим себе, что необходимо составить программу, вычисляющую среднюю температуру за год по известным значениям среднесуточных температур. Можно было бы объявить 365 переменных и найти их среднеарифметическое значение, однако при этом возникли бы трудности с инициализацией и суммированием такого большого количества переменных. Программисту понадобилось бы для этого немало времени и терпения. Гораздо удобнее использовать для этих целей массивы. В программировании существует много базовых алгоритмов, используя которые можно свести любую задачу обработки массивов к компактному коду, состоящему всего из нескольких строк.

4.1 Описание и характеристики одномерных массивов

Массив – это конечная последовательность элементов одного типа, для обращения к которым используется общее имя.

Количество значений в массиве называется его *размером* или *длиной*. Каждому элементу внутри массива соответствует свой номер – *индекс*. Доступ к конкретному элементу массива осуществляется по его индексу. Индексы элементов массива всегда начинаются с нуля. Так, если массив имеет длину $n > 0$, то первый элемент будет иметь индекс – 0, а последний – $n-1$. Индекс последнего элемента в массиве всегда на единицу меньше, чем размер массива (рис.4.1).

<i>Индекс</i>	0	1	2	3	4
<i>Элемент</i>	15.8	-8.28	0.12	-4.125	16.57

Рис.4.1 – Пример одномерного вещественного массива $a[n]$ ($n=5$)

Размер массива задаётся при создании массива и не может быть изменён в дальнейшем. Это означает, что по ходу выполнения программы нельзя удалять элементы из массива или добавлять их туда, однако можно в существующие элементы присвоить новые значения.

Как уже было отмечено выше, массив – это ссылочный тип. Таким образом, имя массива – это ссылка на адрес ячейки памяти, в которой хранится начальный элемент массива. Индекс массива, фактически,

указывает на то, насколько надо отступить от начального элемента массива в памяти, чтоб добраться до нужного элемента.

В качестве элементов массива могут выступать значения любого типа, как примитивные, так и ссылочные, в том числе – переменные типа массив. Тип элемента массива называется *базовым типом* массива.

Описание массива обычно проводится в три этапа.

Первый этап – *объявление*. На этом этапе объявляется имя массива – переменная типа ссылка на массив, содержащая тип массива. Для этого записывается базовый тип массива, квадратными скобками указывается, что объявляется ссылка на массив, а не простая переменная, а далее записывается имя массива.

Возможные следующие варианты объявления массива:

```
тип [] имя;  
тип имя[];
```

где *тип* – это базовый тип элементов массива, а *имя* – имя массива, являющееся уникальным в пределах программы идентификатором, начинающимся с буквы.

В том случае если предполагается объявить сразу несколько имен массивов одного базового типа, предпочтительнее использовать первый вариант объявления. Тогда после указания типа и квадратных скобок списком перечисляются несколько имен массивов, например:

```
double[] a, b;
```

Здесь объявлены имена для двух массивов `a[]` и `b[]`, с которыми далее можно будет связать массивы из вещественных чисел (типа `double`). Обратите внимание, что на этапе объявления массивы еще не созданы, а только подготовлены имена для них.

Второй вариант объявления удобен тогда, когда нужно объявить массив среди определений обычных переменных, например:

```
int i=0, b[], j=1;
```

Второй этап – *определение*. На этом этапе с помощью операции `new` с объявленным ранее именем массива связывают нужный фрагмент памяти, где будут друг за другом (в смежных ячейках) храниться значения элементов массива. Операция `new` выделяет для массива участок в памяти и возвращает в качестве результата адрес этого участка.

На этапе определения после зарезервированного слова `new` указывают имя базового типа массива, а за ним в квадратных скобках число элементов в создаваемом массиве, например, так можно определить массив `a`, состоящий из пяти вещественных элементов:

```
a = new double[5];
```

Можно совместить этапы объявления и определения массива в соответствии со следующей общей формой:

```
тип[] имя = new тип[размер];
```

Например,

```
double[] a = new double[5];  
int i=0, b[] = new int[50], j=1;
```

Третий этап – *инициализация*. На этом этапе элементы массива получают начальные значения.

Начальные значения либо присваиваются элементам массива, путем явного указания их индекса, либо сразу задаются в фигурных скобках через запятую в виде констант или константных выражений.

В первом случае, для того чтобы обратиться к какому-то из элементов массива, нужно указать имя массива и за ним индекс элемента в квадратных скобках, например:

```
a[0] = 15.8;  
a[1] = -8.28;  
a[2] = 0.12;  
a[3] = -4.125;  
a[4] = 16.57;
```

Во втором случае начальные значения задаются в фигурных скобках через запятую. При этом совмещаются этапы определения и инициализации массива:

```
a = {15.8, -8.28, 0.12, -4.125, 16.57};
```

При такой форме записи нет необходимости использовать операцию `new`, а количество элементов массива, будет равно количеству начальных значений. Обратите внимание, что после закрывающей фигурной скобки ставится точка с запятой.

Можно также совместить все три этапа описания массива используя следующую общую форму:

```
тип[] имя = {эл0, эл1, эл2, ..., элN-1};
```

Например,

```
double[] a = {15.8, -8.28, 0.12, -4.125, 16.57};
```

Если массив был создан с помощью операции `new` и не инициализирован, то каждый его элемент получает нулевое значение по умолчанию.

К переменным типа массив можно применять некоторые операции. Например, их можно присваивать друг другу. Так выполнение следующего присваивания:

```
double[] a = new double[5],  
        b = new double[10];  
a=b;
```

приведёт к тому, что ссылочные переменные *a* и *b* будут ссылаться на один и тот же массив, состоящий из 10 вещественных элементов, т.е. содержать один и тот же адрес памяти.

Переменной типа массив можно присвоить значение `null`, не указывающее ни на какой адрес памяти, после чего массив, на который указывала данная ссылка, теряется, если на него не было других ссылок.

```
a = null;
```

Как уже отмечалось выше изменять длину описанного массива нельзя, однако если переменной типа массив был сопоставлен массив определенной длины, то в любой момент этой переменной можно сопоставить новый массив. Например:

```
int[] a = new int[10];  
.  
.  
.  
a = new int[20];
```

При этом прежний массив, находящийся в динамической области памяти, будет утерян.

К переменным типа массив можно применять операции сравнения на равенство: `a==b`, и неравенство: `a!=b`. При этом сравниваются не элементы массивов, а адреса, содержащиеся в ссылках. Таким образом, можно узнать, не ссылаются ли они на один и тот же массив.

Длина каждого массива хранится в поле `length`, которое доступно только для чтения – изменять его путём присваивания нового значения нельзя. Поле содержит целое значение, равное длине массива и для каждого конкретного массива уточняется именем массива через точку. Так, например, для следующих массивов:

```
int[] a = new int[10];  
int n=15;  
double[] b = new double[2*n];  
int[] c={6,3,4,5,6,0,-3,12};
```

поля `length` примут значения: `a.length=10`, `b.length=30`, `c.length=8`.

4.2 Базовые алгоритмы обработки одномерных массивов

Любая обработка массива предусматривает выполнение определенных операций над его элементами. Рассмотрим простую, но очень важную операцию – выбор элемента массива.

Для доступа к определенному элементу массива используется операция *индексирования* [], общая форма которой имеет следующий вид:

имя [индексное выражение]

Индексом массива может служить целочисленное значение типа `byte`, `short`, `int` или `char`, за исключением типа `long`. Индексы можно задавать любыми целочисленными выражениями, кроме типа `long`, например, `a[2*i]`, `a[i%5]`, `a[++i]`.

Исполняющая система Java следит за тем, чтобы значения индексных выражений не выходили за границы длины массива. Это значит, что любой доступ к массиву влечет проверку, не выходит ли индекс массива за пределы своего диапазона. Если произошел выход за границы допустимого диапазона, возникает исключительная ситуация (*выход индекса за границу массива*).

При выборе всех элементов массива приходится повторять однотипные действия. Для многократного повторения операций над элементами массива используют циклы. Наиболее приемлемым вариантом является применение цикла с параметром `for`. Обычно параметр цикла задает изменение значения индекса элементов в диапазоне `[0, length-1]`. Например, следующий код программы:

```
int [] a = new int [5];
for(int i=0; i<a.length; i++) {
    a[i] = (i+1) * 10; }
```

выполняет присваивания начальных значений элементам массива и заменяет присваивания вида:

```
a[0]=10; a[1]=20; a[2]=30; a[3]=40; a[4]=50;
```

Понятно, что для массива, состоящего из большего количества элементов, например, из 100 элементов, применение цикла в подобном случае является единственным возможным решением.

4.2.1 Ввод и вывод массива

Ввод и вывод массивов выполняется поэлементно с помощью оператора цикла с заданным числом повторений, в котором количество изменений параметра цикла соответствует максимально возможному количеству элементов массива (рис.4.2). При организации ввода и вывода массива следует помнить, что количество и тип элементов массива задаются при его описании и не могут быть изменены, а превышение верхней границы длины массива может привести к ошибкам.

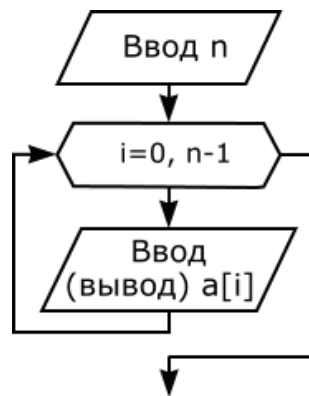


Рис.4.2 – Блок-схемы алгоритма ввода-вывода одномерного массива

Ввод элементов массива можно осуществлять с клавиатуры, тогда в программе необходимо организовать цикл, в каждой итерации которого будет вводиться начальное значение, тип которого соответствует базовому типу массива, и присваиваться текущему элементу:

```

Scanner scan = new Scanner(System.in);
System.out.println("Введите количество элементов n:");
int n= scan.nextInt();
int[] a = new int[n];
for (int i=0;i<a.length;i++){
    System.out.printf("Введите a[%d]=", i+1);
    a[i]=scan.nextInt();}
  
```

Как видно, в примере количество элементов массива n вводится с клавиатуры пользователем, это делает программу более универсальной.

В отдельных случаях, например на этапе тестирования программы, удобно организовать инициализацию массива с помощью генератора псевдослучайных чисел:

```

for(int i=0;i<a.length;i++)
    a[i]=Math.floor(Math.random()*10);
  
```

В примере элементы массива $a[]$ инициализируются целыми случайными числами в диапазоне от 0 до 9.

Вывод элементов массива на экран также организуют в цикле, причем вывести элементы можно либо в строку, либо в столбец. Для вывода элементов массива в столбец используют управляющий символ перевода строки ' $\backslash n$ ' или метод `println()`.

– вывод в столбец

```

for (int i=0;i<a.length;i++)
    System.out.printf("%n%d", a[i]);
  
```

– вывод в строку

```

for (int i=0;i<a.length;i++)
  
```

```
System.out.printf("  %d", a[i]);
```

Задача 4.1. Создайте одномерный массив, состоящий из 100 вещественных чисел. Инициализируйте элементы массива случайными числами в диапазоне от -10.0 до 10.0. Организуйте вывод элементов массива на экран по 5 элементов в строке.

```
public class Ex_4_1 {
    public static void main (String[] args){
        int n=5;
        double[] a = new double[100];
        for (int i=0;i<a.length;i++){
            a[i]=20*Math.random()-10;
            System.out.printf("  a[%2d]=%5.1f", i+1, a[i]);
            if((i+1)%n==0)
                System.out.println();//перевод строки
        }
    }
}
```

Результат:

```
a[ 1]=  8,9 a[ 2]=  1,7 a[ 3]= -3,0 a[ 4]= -1,1 a[ 5]= -7,1
a[ 6]= -8,1 a[ 7]=  5,1 a[ 8]= -2,3 a[ 9]=  8,9 a[10]=  7,6
a[11]= -4,0 a[12]=  1,0 a[13]=  0,0 a[14]=  7,4 a[15]= -9,4
a[16]= -5,0 a[17]=  0,8 a[18]= -6,7 a[19]= -8,9 a[20]=  9,6
a[21]=  1,4 a[22]= -6,1 a[23]= -0,0 a[24]=  5,4 a[25]=  2,9
a[26]= -9,4 a[27]= -2,2 a[28]=  9,4 a[29]=  2,6 a[30]= -5,8
a[31]= -8,0 a[32]= -4,3 a[33]= -2,6 a[34]= -9,0 a[35]=  1,5
a[36]=  4,2 a[37]= -2,9 a[38]=  6,5 a[39]= -7,1 a[40]= -7,6
a[41]=  1,9 a[42]= -4,3 a[43]= -5,7 a[44]=  0,3 a[45]=  0,9
a[46]=  0,1 a[47]= -0,6 a[48]= -4,4 a[49]=  5,7 a[50]=  2,6
a[51]=  1,4 a[52]= -2,1 a[53]= -7,6 a[54]=  7,0 a[55]= -0,5
a[56]= -6,2 a[57]= -1,0 a[58]=  4,5 a[59]=  1,5 a[60]= -7,3
a[61]= -0,4 a[62]= -4,1 a[63]= 10,0 a[64]= -6,8 a[65]=  9,2
a[66]= -3,9 a[67]= -8,3 a[68]= -7,7 a[69]=  0,2 a[70]= -0,2
a[71]= -7,5 a[72]= -1,2 a[73]=  2,9 a[74]= -3,7 a[75]= -9,3
a[76]= -2,3 a[77]=  5,5 a[78]= -5,9 a[79]= -5,6 a[80]= -9,8
a[81]= -4,5 a[82]= -6,6 a[83]=  1,7 a[84]=  5,2 a[85]=  4,3
a[86]=  7,0 a[87]=  8,3 a[88]=  0,2 a[89]= -0,2 a[90]=  6,9
a[91]= -2,8 a[92]= -9,9 a[93]=  8,0 a[94]=  8,8 a[95]=  6,4
a[96]=  9,1 a[97]=  3,9 a[98]=  6,7 a[99]= -1,7 a[100]= -3,7
```

4.2.2 Версия for-each цикла for

Для выполнения строго последовательных повторяющихся действий над элементами массива можно использовать усовершенствованный вариант цикла for – версию for-each.

Версия цикла for-each предназначена для работы с различными типами коллекций объектов, к которым относятся и массивы. В общем

случае коллекции в Java – это набор готовых классов, реализующих механизмы работы со структурами данных. В отличие от массивов коллекции поддерживают ряд важных дополнительных операций, например, динамическое изменение длины.

Общая форма версии `for-each` цикла `for` для массивов имеет следующий вид:

```
for(тип итер_пер : имя) {тело цикла}
```

где `тип` – базовый тип массива;

`итер_пер` – имя итерационной переменной, которая последовательно будет принимать значения элементов массива, от первого и до последнего;

`имя` – имя массива, по которому должен выполняться цикл.

На каждой итерации цикла `for-each` извлекается следующий элемент массива, который сохраняется в переменной `итер_пер`. Тип итерационной переменной должен быть совместим с типом элементов массива. Цикл выполняется до тех пор, пока не будут перебраны все элементы массива.

Рассмотрим, как изменятся циклы для вывода элементов массива на экран при использовании версии `for-each`.

<i>Цикл for</i>	<i>Цикл for-each</i>
вывод в столбец	
<pre>for (int i=0;i<a.length;i++) System.out.printf("%n%d",a[i]);</pre>	<pre>for (int x : a) System.out.printf("%n%d",x);</pre>
вывод в строку	
<pre>for (int i=0;i<a.length;i++) System.out.printf(" %d",a[i]);</pre>	<pre>for (int x : a) System.out.printf(" %d",x);</pre>

Как видно, применение версии `for-each` цикла `for` позволяет не устанавливать значение параметра цикла за счет указания его начального и конечного значений и исключает необходимость индексации массива вручную. Цикл автоматически последовательно перебирает все элементы массива.

Версию `for-each` цикла `for` удобно использовать в алгоритмах, основанных на последовательном переборе элементов массива, начиная с первого и заканчивая последним. К таким алгоритмам можно отнести, например, поиск элемента или расчет суммы элементов массива. Однако

применение цикла `for-each` ограничено и подходит не для всех задач. Так следует помнить, что итерационная переменная является переменной “только для чтения”, поэтому оператор присваивания значения итерационной переменной не оказывает никакого влияния на исходный массив. Иначе говоря, содержимое массива нельзя изменять, присваивая новое значение итерационной переменной. С учетом этого, ввод и инициализацию элементов массива следует проводить с помощью обычного цикла `for`, как это было показано в предыдущем параграфе.

В любом случае, в дальнейшем при рассмотрении базовых алгоритмов обработки одномерных массивов будем отдельно отмечать алгоритмы, в которых применение версии `for-each` цикла `for` является обоснованным.

4.2.3 Вычисление сумм, количеств и произведений элементов массива

Сумма элементов одномерного массива $a[n]$ вычисляется в цикле с параметром путем прибавления каждого очередного элемента массива к сумме предыдущих элементов. При этом используется оператор присваивания для накопления суммы: `sum=sum+a[i]`. Он указывает на необходимость прибавления к уже известному значению суммы (переменная `sum` в правой части арифметического выражения) значения очередного слагаемого $a[i]$ и присваивания полученного значения в качестве нового значения переменной `sum`. Выражение можно переписать, с использованием составного оператора присваивания: `sum+=a[i]`.

В процессе вычисления суммы при первом прохождении цикла первоначальное значение переменной `sum` прибавляется к значению нулевого элемента массива $a[0]$, следовательно, начальное значение `sum` не должно влиять на конечный результат, что возможно, только если оно равно 0. Поэтому перед выполнением цикла переменной `sum` присваивается нулевое значение. Блок-схема вычисления суммы и произведения элементов одномерного массива представлена на рис.4.3.

Как видно из рис.4.3, произведение элементов массива накапливается по аналогичной формуле `pr*=a[i]`, при обязательном присвоении переменной `pr` начального значения, равного 1 (`pr=1`).

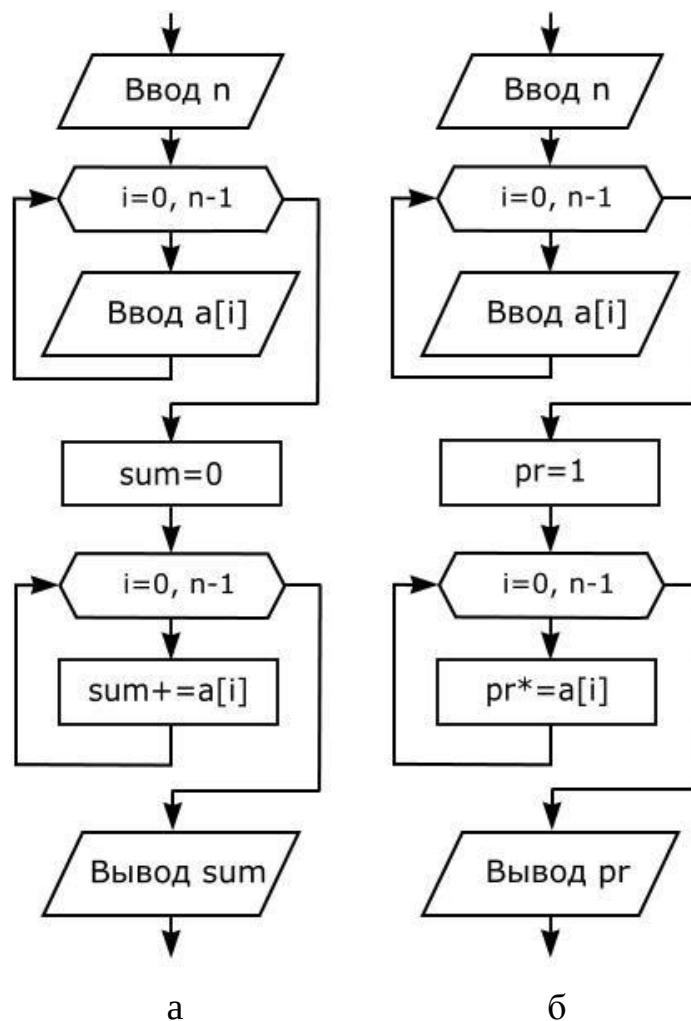


Рис. 4.3 – Блок-схема алгоритма вычисления суммы (а) и произведения (б) элементов одномерного массива

Задача 4.2. Составьте программу нахождения суммы и произведения всех элементов вещественного одномерного массива.

Объяснение: для хранения суммы и произведения объявлены переменные `sum` и `pr`, которые имеют тот же тип данных, что и элементы массива. В программы в блоке комментариев приведен альтернативный код нахождения суммы и произведения элементов массива с помощью версии `for-each` цикла `for`.

```

import java.util.Scanner;
public class Ex_4_2 {
    public static void main (String[] args){
        Scanner scan = new Scanner(System.in);
        System.out.println("Введите n:");
        int n=scan.nextInt();

```

```

double [] a = new double[n];

//Ввод элементов массива
for (int i=0;i<a.length;i++){
    System.out.printf("Введите элемент a[%d]:", i+1);
    a[i]=scan.nextDouble();}

//нахождение суммы и произведения
double sum=0, pr=1;
for (int i=0;i<a.length;i++){
    sum+=a[i]; pr*=a[i]; }

/* нахождение суммы и произведения с помощью
циклаfor-each
for (double x : a)
{sum+=x; pr*=x;} */

System.out.printf("Сумма = %5.2f\n", sum);
System.out.printf("Произведение = %5.2f", pr);
}}

```

Результат:

```

Введите n:
5
Введите элемент a[1]:2,5
Введите элемент a[2]:-6,4
Введите элемент a[3]:3,6
Введите элемент a[4]:-1,2
Введите элемент a[5]:0,3
Сумма = -1,20
Произведение = 20,74

```

Для подсчета количества элементов массива, удовлетворяющих определенному условию, удобно использовать целочисленную переменную-счетчик `counter` с нулевым начальным значением. В цикле с параметром следует перебрать все элементы массива, последовательно проверяя, удовлетворяет ли условию каждый элемент массива. В случае нахождения элемента, удовлетворяющего условию, переменную-счетчик следует увеличить на 1 (`counter++`). После завершения цикла переменная-счетчик будет содержать количество элементов, удовлетворяющих некоторому условию. Блок-схема алгоритма вычисления количества элементов одномерного массива, удовлетворяющих условию представлена на рис.4.4.

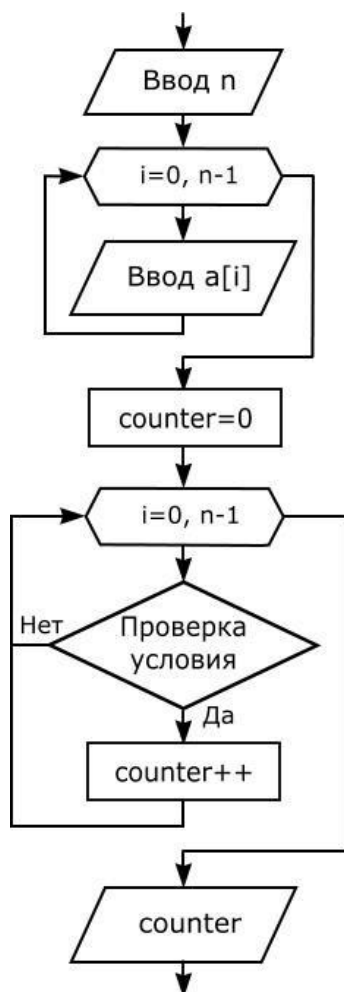


Рис. 4.4 – Блок-схема алгоритма вычисления количества элементов одномерного массива, удовлетворяющих условию

Действия, выполняемые в рассмотренных алгоритмах вычисления суммы, произведения и количества элементов, удовлетворяющих некоторому условию, сведены в табл.4.1.

Таблица 4.1 – Действия, выполняемые при нахождении суммы, произведения и количества элементов одномерного массива, удовлетворяющих условию

<i>Характеристика массива</i>	<i>До цикла</i>	<i>В цикле</i>	<i>После цикла</i>
сумма	sum=0;	sum+=a[i];	Вывод sum
произведение	pr=1;	pr*=a[i];	Вывод pr
количество	counter=0;	if (условие) counter++;	Вывод counter

При поиске элементов, удовлетворяющих определенному условию, или при нахождении их количества, в программе приходится каждый раз проверять соответствует ли текущий элемент заданному условию или свойству: четности-нечетности, положительности-отрицательности и т.п. Запись некоторых, наиболее часто встречающихся в задачах свойств элементов приведена в табл.4.2.

Таблица 4.2 – Запись некоторых свойств элементов массива

<i>Свойство</i>	<i>Запись условия</i>
четные	<code>a[i]%2==0</code>
нечетные	<code>a[i]%2!=0</code>
кратные k	<code>a[i]%k==0</code>
некратные k	<code>a[i]%k!=0</code>
на четных местах	<code>i%2==0</code>
на нечетных местах	<code>i%2!=0</code>
положительные	<code>a[i]>0</code>
неотрицательные	<code>a[i]>=0</code>
отрицательные	<code>a[i]<0</code>
меньше x или больше y	<code>a[i]<x a[i]>y</code>
в интервале (x,y)	<code>a[i]>x&&a[i]<y</code>
в интервале [x,y]	<code>a[i]>=x&&a[i]<=y</code>
запись числа заканчивается на цифру k	<code>a[i]%10==k</code>

Задача 4.3. Известны результаты экзамена 25 студентов одной группы по программированию. Определить, сколько студентов сдали экзамен на 4 и 5.

Объяснение: В данной программе список оценок хранится в массиве `a[]`, количество студентов сдавших экзамен на 4 и 5 – в переменной `counter`. Программа предусматривает проверку корректности вводимых данных: при попытке ввода несуществующей по пятибалльной системе оценки, программа повторяет ее ввод. Для этого используется цикл с постусловием `do-while`.

```
import java.util.Scanner;
public class Ex_4_3 {
    public static void main (String[] args){
```



```

Scanner scan = new Scanner(System.in);
int [] a = new int[25];
int counter=0;

for (int i=0;i<a.length;i++){

    //проверка корректности вводимых данных
    do{
System.out.printf("Введите оценку %d-го студента:", i+1);
        a[i]=scan.nextInt();
    } while (a[i]<1||a[i]>5);

    if (a[i]>3) counter++; }
    System.out.printf("На 4 и 5 сдали экзамен %d студентов",
counter);
    }}

```

На основе комбинации рассмотренных типовых алгоритмов вычисления суммы, количества и произведения элементов массива могут быть построены другие алгоритмы вычисления характеристик массива. Например, алгоритмы решения задач вычисления среднего арифметического и среднего геометрического элементов массива, удовлетворяющих некоторому условию. Напомним, что среднее арифметическое значение определяется как сумма всех элементов массива, деленная на их количество. Для нахождения среднего геометрического нужно перемножить все элементы массива и извлечь из них корень, степень которого равна количеству элементов.

Для вычисления среднего арифметического необходимо найти сумму и количество элементов, удовлетворяющих поставленному условию. А для вычисления среднего геометрического надо найти произведение и количество нужных элементов. Рассмотрим особенности программной реализации этих алгоритмов на примере задачи вычисления среднего арифметического четных элементов и среднего геометрического нечетных элементов массива.

Задача 4.4. Для массива, содержащего n целых положительных чисел, определите среднее арифметическое четных элементов и среднее геометрическое нечетных элементов.

Объяснение: в программе объявлены следующие переменные:

$a[]$ – заданный массив.

n – количество элементов массива.

sum и pr – переменные для хранения суммы и произведения, они всегда имеют тот же тип данных, что и элементы массива.

counter1 и counter2 – переменные для хранения количества четных и нечетных элементов массива, соответственно, имеют целый тип.

В качестве начальных значений заданы: ноль для суммы и количеств и единица для произведения.

Значения среднего арифметического и среднего геометрического (arifm и geom) не сохраняются в отдельных переменных, а вычисляются при выводе на экран в методе printf(). Так как значения вычисляются в результате деления и извлечения корня некоторой степени, то они всегда имеют вещественный тип. Кроме того в Java результат деления целых чисел также является целым числом, поэтому при расчете данных характеристик применяется явное преобразование типов.

В программе осуществляется проверка на четность. Так как четные числа делятся на 2 без остатка, то в качестве условия четности можно использовать логическое выражение: $a[i] \% 2 == 0$.

В программе там, где это возможно использована версия for-each цикла for.

```
import java.util.Scanner;
public class Ex_4_4 {
    public static void main (String[] args){
        Scanner scan = new Scanner(System.in);

        System.out.println("Введите n:");
        int n=scan.nextInt();
        int [] a = new int[n];
        //Ввод элементов массива
        for (int i=0;i<a.length;i++){
            System.out.printf("Введите элемент a[%d]:", i+1);
            a[i]=scan.nextInt();}

        //Вывод исходного массива на экран
        System.out.println("Исходный массив");
        for(int x : a) System.out.printf("%d\n", x);

        //Вычисление суммы, произведения и количеств
        //элементов с учетом условия четности элементов
        int sum=0, pr=1, counter1=0, counter2=0;
        for(int x : a)
            if (x%2==0) {counter1++; sum+=x;}
                else {counter2++; pr*=x;}

        //проверки на равенства нулю количеств элементов
        //и вычисление среднего арифметического и среднего
        //геометрического значений
        if (counter1==0)
            System.out.printf("Четных элементов нет\n");
        else
            System.out.printf("Arifm=%5.2f\n", (double)sum/counter1);
```

```

        if (counter2==0)
            System.out.printf("Нечетных элементов нет%n");
        else
            System.out.printf("Geom= %5.2f", Math.pow(pr,1./counter2));
    }}

```

Результат:

Введите количество элементов массива:

5

Введите элемент a[1]:5

Введите элемент a[2]:6

Введите элемент a[3]:7

Введите элемент a[4]:8

Введите элемент a[5]:9

Исходный массив

5

6

7

8

9

Arifm= 7,00

Geom= 6,80

4.2.4 Поиск максимального и минимального элемента массива

Для нахождения максимального (минимального) элемента массива некоторой целевой переменной (назовем ее `max` для максимального и `min` для минимального элемента) присваивается значение *нулевого элемента массива*. В цикле, начиная с первого элемента, сравнивается очередной элемент массива с целевой переменной. Если текущий элемент массива больше (при поиске максимального элемента) или меньше (при поиске минимального элемента) переменной `max` или `min` соответственно, то целевой переменной присваивается значение этого текущего элемента массива, в противном случае значение переменной не изменяется. После завершения цикла целевая переменная будет содержать значение максимального (минимального) элемента массива.

При поиске номера максимального (минимального) элемента возникает вторая целевая переменная – номер искомого элемента. Назовем ее `nommax` для номера максимального и `nommin` для номера минимального элемента. До начала цикла этой переменной должно быть присвоено значение 0. В цикле переменная, хранящая номер, меняется вместе с переменной, хранящей значение максимального (минимального) элемента.

Если необходимо найти номер первого максимального (минимального) элемента, то используется операция «больше» («меньше»). Если необходимо найти номер последнего максимального (минимального) элемента, то используется операция «больше или равно»

(«меньше или равно»). На рис.4.5 представлены блок-схемы поиска максимального и минимального элемента массива.

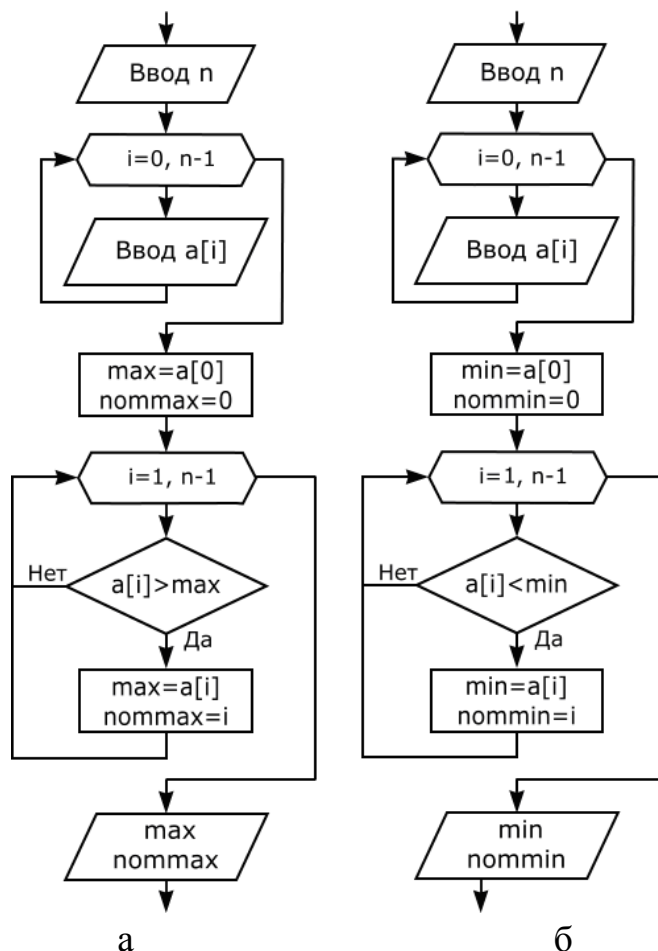


Рис.4.5 – Блок-схемы поиска максимального элемента массива и его номера (а) и минимального элемента массива и его номера (б).

Действия, выполняемые в рассмотренных алгоритмах поиска максимального и минимального элемента и их номеров в целочисленном и вещественном массивах сведены в табл.4.3.

Таблица 4.3. Действия, выполняемые при поиске максимального и минимального элемента массива и их номеров

<i>До цикла</i>	<i>В цикле</i>	<i>После цикла</i>
<i>Поиск максимального элемента</i>		
<code>max=a[0];</code>	<code>for(int i=1;i<a.length;i++) if (a[i]>max) max=a[i];</code>	Вывод max

<i>Поиск номера максимального элемента</i>		
nommax=0;	for(int i=1;i<a.length;i++) if (a[i]>a[nommax]) nommax=i;	Вывод nommax
<i>Поиск минимального элемента</i>		
min=a[0];	for(int i=1;i<a.length;i++) if (a[i]<min) min=a[i];	Вывод min
<i>Поиск номера минимального элемента</i>		
nommin=0;	for(int i=1;i<a.length;i++) if (a[i]<a[nommin]) nommin=i;	Вывод nommin

Рассмотрим некоторых задачи, для решения которых требуется найти минимальный (максимальный) элемент массива и его номер.

Задача 4.5. Для целочисленного массива, содержащего n элементов, определите, сколько положительных элементов располагается между максимальным и минимальными элементами.

Объяснение: задача сводится к поиску порядковых номеров максимального и минимального элементов. Порядок расположения элементов в массиве заранее не известен – сначала может следовать как максимальный, так и минимальный элемент, более того, они могут совпадать. Поэтому прежде чем искать количество положительных элементов, требуется определить, какой из этих индексов больше, чтобы просматривать массив от меньшего номера к большему. Для определения количества положительных элементов объявим переменную-счетчик, которая будет увеличивать свое значение на 1 всякий раз, когда будет найден положительный элемент. В программе используются следующие переменные:

n – количество элементов;
 $a[]$ – исходный массив;
 $nommin$ – номер минимального элемента;
 $nommax$ – номер максимального элемента;
 $i1$ – левая граница интервала поиска положительных элементов;
 $i2$ – правая граница интервала поиска положительных элементов;
 $counter$ – количество положительных элементов.

В программе не учитывается возможность нахождения в массиве нескольких максимальных и минимальных элементов.

```
import java.util.Scanner;
public class Ex_4_5 {
    public static void main (String[] args){
        Scanner scan = new Scanner(System.in);
        System.out.println("Введите n:");
```

```

int n=scan.nextInt();
int [] a = new int[n];
//Ввод элементов массива
for (int i=0;i<a.length;i++){
System.out.printf("Введите элемент a[%d]:", i+1);
a[i]=scan.nextInt();}

//Вывод исходного массива на экран
System.out.println("Исходный массив");
for(int x : a) System.out.printf("%d\n", x);
int nommax=0, nommin=0, counter=0;

//Поиск номера минимума и максимума
for(int i=1; i<a.length; i++) {
if (a[i]<a[nommin]) nommin=i;
if (a[i]>a[nommax]) nommax=i;
}
// Определение границ интервала поиска
int i1=nommin<nommax?nommin+1:nommax+1;
int i2=nommin<nommax?nommax:nommin;

//Поиск положительных элементов
for (int i=i1;i<i2;i++)
    if(a[i]>0) counter++;
System.out.printf("counter= %d", counter);
}}

```

Результат:

```

Введите n:
6
Введите элемент a[1]:5
Введите элемент a[2]:15
Введите элемент a[3]:-8
Введите элемент a[4]:-4
Введите элемент a[5]:5
Введите элемент a[6]:-15
Исходный массив
5
15
-8
-4
5
-15
counter= 1

```

Задача 4.6. Ежедневные замеры температуры воздуха в течение декабря дали следующие результаты: t_1, t_2, K, t_{31} . Определите наименьшую наблюдавшуюся температуру и дату, когда она была зарегистрирована в первый раз. Определите наибольшую наблюдавшуюся температуру и дату, когда она была зарегистрирована в последний раз.

Объяснение: в программе используются следующие переменные:

$t[]$ – массив температур;

max – максимальная температура;

nommax – номер последнего в массиве максимального элемента;

min – минимальная температура;

nommin – номер первого в массиве минимального элемента.

```
import java.util.Scanner;
public class Ex_4_6 {
    public static void main (String[] args){
        Scanner scan = new Scanner(System.in);
        double [] t = new double [31];
        //Ввод элементов массива
        for (int i=0;i<t.length;i++){
            System.out.printf("Введите элемент t[%d]:", i+1);
            t[i]=scan.nextDouble();
        }

        //Вывод исходного массива на экран
        System.out.println("Исходный массив температур");
        for (int i=0;i<t.length;i++)
            System.out.printf("%d декабря->%5.1f%n", i+1, t[i]);

        double min=t[0], max=t[0];
        int nommax=0, nommin=0;

        //Поиск минимума и максимума
        for(int i=1; i<t.length; i++) {
            if (t[i]<min) {min=t[i]; nommin=i;}
            if (t[i]>=max) {max=t[i]; nommax=i;}
        }
        System.out.printf("Min=%5.1fград., дата - %d декабря%n",
min, nommin+1);
        System.out.printf("Max=%5.1fград., дата - %d декабря",
max, nommax+1);
    }
}
```

Фрагмент результата:

Исходный массив температур

```
1 декабря-> -9,0
2 декабря-> -5,0
3 декабря-> -9,0
4 декабря-> -8,0
5 декабря-> 0,0
6 декабря-> 0,0
7 декабря-> -4,0
8 декабря-> 0,0
9 декабря-> -3,0
10 декабря-> -3,0
11 декабря-> -4,0
. . .
```

```
28 декабря-> -1,0
29 декабря-> -8,0
30 декабря-> -9,0
31 декабря-> -5,0
```

```
Min= -9,0град., дата - 1 декабря
Max= 0,0град., дата - 8 декабря
```

4.2.5 Поиск элементов в массиве

При решении задачи 4.6 нам пришлось организовывать поиск первого минимального и последнего максимального элементов массива. Рассмотрим подробнее алгоритмы поиска первого и последнего элементов, удовлетворяющих некоторому условию. Это особенно актуально тогда, когда таких элементов в массиве несколько.

Поиск предполагает определение номера или значения элемента, удовлетворяющего некоторому условию. Поиск проводят слева направо, от нулевого элемента к последнему элементу. При этом нельзя исключать ситуацию, что в массиве вообще нет искомых элементов. Поэтому если нужно найти номер элемента, то сначала он задается равным -1. Затем последовательно проверяются все элементы массива в цикле. Если определяется первый элемент, удовлетворяющий условию, то его номер или значение запоминается в специальной переменной, а сам цикл немедленно прерывается. Так как после обнаружения первого искомого элемента анализ других элементов массива является лишним. Отметим, что поиск, предполагающий последовательный просмотр всех элементов массива называется *линейным*.

Проиллюстрируем данный алгоритм фрагментом программы, в котором организуем поиск номера первого элемента массива, равного нулю:

```
int nom=-1;
for(int i=0; i<a.length; i++)
if(a[i]==0) {nom=i; break;}
if (nom==-1) System.out.print("В массиве нет нулей");
    else System.out.print(nom);
```

Еще проще выглядит цикл, когда нужно найти и вывести на экран первый элемент массива, равный нулю:

```
for(int i=0; i<a.length; i++)
if(a[i]==0) {System.out.print(a[i]); break;}
```

Если необходимо вывести все элементы равные нулю, то прерывание выполнять не нужно, а цикл следует выполнять до последнего элемента:

```
for(int i=0; i<a.length; i++)
if(a[i]==0) System.out.print(a[i]);
```

Задача поиска последнего элемента, удовлетворяющего некоторому условию, имеет два разных решения. Первый способ предлагает анализировать массив от нулевого элемента к последнему. Как только в нем встречается очередной элемент, удовлетворяющий поставленному условию, его номер запоминается в специальной переменной. Но в отличие от предыдущей задачи цикл не прерывается, а продолжается дальше. Так как все номера последовательно записываются в одну и ту же переменную, то после завершения цикла в ней будет храниться номер

последнего элемента, удовлетворяющего заданному условию. В соответствии с данным алгоритмом запишем фрагмент программы поиска последнего элемента массива, равного нулю:

```
int nom=-1;
for(int i=0; i<a.length; i++)
if(a[i]==0) nom=i;
if (nom==-1) System.out.print("В массиве нет нулей");
    else System.out.print(nom);
```

Второй способ предполагает анализ массива в обратном направлении: от последнего элемента к нулевому. В этом случае первый элемент, удовлетворяющий поставленному условию, и будет искомым элементом. Достаточно запомнить его номер и прервать цикл.

```
int nom=-1;
for(int i=a.length-1; i>=0; i--)
if(a[i]==0) {nom=i; break;}
if (nom==-1) System.out.print("В массиве нет нулей");
    else System.out.print(nom);
```

4.2.6 Перестановка элементов массива

Задача перестановки или обмена значениями между двумя элементами массива решается тем же способом, что и задача обмена значений двух переменных. Для обмена необходима дополнительная переменная, в которой временно будет храниться значение одного из переставляемых элементов. Алгоритм можно представить следующими шагами:

- 1) Значение одного элемента массива сохраняется в дополнительной переменной.
- 2) Значение другого элемента присваивается первому элементу.
- 3) Значение дополнительной переменной присваивается второму элементу.

Задача 4.7. *Переставьте местами два элемента массива, номера которых вводятся с клавиатуры.*

Объяснение: перед перестановкой необходимо проверить, существуют ли в массиве элементы с такими номерами, не выходят ли эти номера за границы массива. Если перестановка возможна, то она проводится с использованием дополнительной переменной *temp*. Массив выводится дважды: до и после перестановки элементов, чтобы пользователь мог проконтролировать правильность работы программы.

```
import java.util.Scanner;
public class Ex_4_7 {
    public static void main (String[] args){
```

```

Scanner scan = new Scanner(System.in);
System.out.println("Введите n:");
int n=scan.nextInt();
int temp, n1, n2; //индексы элементов
int [] a = new int[n];

//Ввод элементов массива
for (int i=0;i<a.length;i++){
    System.out.printf("Введите элемент a[%d]:", i+1);
    a[i]=scan.nextInt();}

System.out.println("Исходный массив:");
for (int x : a) System.out.printf("%d  ", x);

//проверка корректности введенных индексов
do{
    System.out.printf("%nВведите индекс первого элемента в
интервале (0;%d):",n-1);
    n1=scan.nextInt();
    }while(n1<0||n1>=n);

do{
    System.out.printf("Введите индекс второго элемента в
интервале (0;%d):",n-1);
    n2=scan.nextInt();
    }while(n2<0||n2>=n);

//Обмен элементов
temp = a[n1]; a[n1]=a[n2]; a[n2]=temp;
System.out.println("Новый массив:");
for (int x : a) System.out.printf("%d  ", x);
}}

```

Результат:

```

Введите n:
5
Введите элемент a[1]:6
Введите элемент a[2]:8
Введите элемент a[3]:1
Введите элемент a[4]:2
Введите элемент a[5]:7
Исходный массив:
6 8 1 2 7
Введите индекс первого элемента в интервале (0;4):2
Введите индекс второго элемента в интервале (0;4):7
Введите индекс второго элемента в интервале (0;4):4
Новый массив:
6 8 7 2 1

```

На основе базового алгоритма перестановки элементов массива могут быть решены разнообразные задачи. Одна из них – инвертирование массива, которая предусматривает перестановку элементов массива в

обратном порядке: нулевой с последним, первый с предпоследним, и т.д. Не трудно вычислить закономерность: i -ый элемент меняется местами с $(n-i-1)$ -ым элементом. Количество таких перестановок в два раза меньше количества элементов.

Задача 4.8. Заполните одномерный целочисленный массив случайными целыми двузначными положительными четными числами. Переставьте элементы массива в обратном порядке.

```
import java.util.Scanner;
public class Ex_4_8 {
    public static void main (String[] args){
        Scanner scan = new Scanner(System.in);

        System.out.println("Введите n:");
        int n=scan.nextInt();
        int temp;
        int [] a = new int[n];

        //заполнение случайными числами
        for(int i=0;i<a.length;i++)
            a[i]=2*((int) (Math.random()*45)+5);

        System.out.println("Исходный массив:");
        for (int x : a) System.out.printf("%d  ", x);

        //перестановка
        for(int i=0;i<a.length/2;i++)
        {temp = a[i];
            a[i]=a[n-i-1];
            a[n-i-1]=temp;}

        System.out.printf("\nНовый массив:\n");
        for (int x : a) System.out.printf("%d  ", x);
    }
}
```

Результат:

```
Введите n:
10
Исходный массив:
18 46 92 44 54 34 60 64 90 22
Новый массив:
22 90 64 60 34 54 44 92 46 18
```

Краткое описание некоторых других задач алгоритмы решения, которых основаны на операции перестановки элементов массива, представлены в табл.4.4.

Таблица 4.4 – Краткое описание алгоритмов решения задач, требующих перестановок элементов массива

<i>Метод решения</i>	<i>Фрагмент кода</i>
<i>Задача 1. Перестановка соседних элементов массива</i>	
Элементы массива переставляются попарно - нулевой со первым, второй с третьим, т.е. элемент с нечетным номером с элементом, имеющим четный номер. $(2*i+1)$-ый элемент меняем местами с $(2*i)$-ым. Количество перестановок в два раза меньше количества элементов. Исходный массив: 16 54 66 66 34 76 Новый массив: 54 16 66 66 76 34	<pre>for(int i=0;i<a.length/2;i++) {temp = a[2*i+1]; a[2*i+1]=a[2*i]; a[2*i]=temp;}</pre>
<i>Задача 2. Обмен половин массива</i>	
Переставляются элементы массива попарно - i-ый с $(i+k)$-ым, где $k=n/2$. Количество таких перестановок в два раза меньше количества элементов. Исходный массив: 52 94 40 22 76 70 Новый массив: 22 76 70 52 94 40	<pre>k=a.length/2; for(int i=0;i<k;i++) {temp = a[i]; a[i]=a[i+k]; a[i+k]=temp;}</pre>
<i>Задача 3. Перестановка максимального и минимального элементов</i>	
Определяются номера максимального и минимального элементов, и обмениваются элементы с соответствующими номерами. Исходный массив: 10 64 70 28 26 46 Новый массив: 70 64 10 28 26 46	<pre>nommax=0; nommin=0; for(int i=1;i< a.length;i++){ if (a[i]>a[nommax]) nommax=i; if (a[i]<a[nommin]) nommin=i; } temp=a[nommin]; a[nommin]=a[nommax]; a[nommax]=temp;</pre>

4.2.7 Вставка, удаление и циклический сдвиг элементов массива

При работе с массивом можно вставлять новый элемент в заданную позицию или удалять заданный элемент.

Для того чтобы вставить новый элемент в заданную позицию k нужно освободить под него место в массиве. Для этого перемещают элементы массива, начиная с k -ой позиции вправо на одну позицию, так как это показано на рис. 4.6. А затем элементу с индексом k присваивают новое значение.

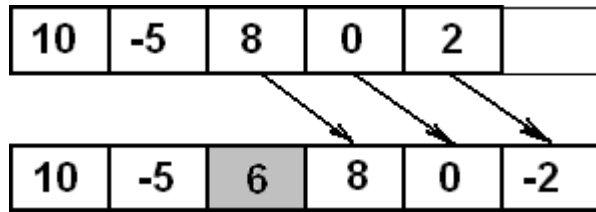


Рис. 4.6 – Вставка элемента в массив

Обратите внимание на то, что в результате вставки количество элементов массива увеличивается, поэтому нужно следить за тем, чтобы не выйти за границы той размерности, которая была указана при объявлении массива. Ниже приведен фрагмент программного кода, который в массив `a[]` вставляет новый элемент `num` в заданную позицию `k`, `n` – количество элементов в массиве.

```
for(int i=n-1; i<=k; i--)
a[i+1]=a[i];
a[k]=num;
n++;
```

Чтобы удалить k -ый элемент массива необходимо сдвинуть все последующие элементы, начиная с $k+1$ на одну позицию влево (к началу массива), чтобы заполнить пустой элемент. Пример показан на рис.4.7. Следует также учесть, что количество элементов массива при этом уменьшится.

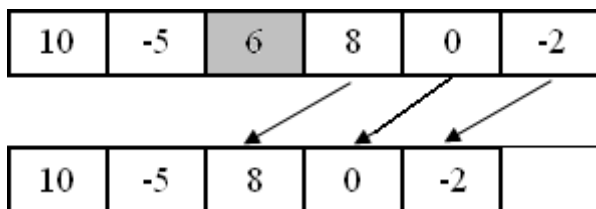


Рис.4.7 – Удаление элемента массива

Ниже приведен фрагмент программного кода, который в массиве `a[]` удаляет k -ый элемент, `n` – количество элементов в массиве.

```
for(int i=k; i<n-1; i++)
a[i]=a[i+1];
a[k]=num;
n--;
```

Элементы массива можно циклически сдвигать влево или вправо (рис.4.8) при этом размер массива не изменяется.

10	-5	8	0	-2
-2	10	-5	8	0

а

10	-5	8	0	-2
-5	8	0	-2	10

б

Рис 4.8 – Циклический сдвиг элементов массива на одну позицию вправо (а) и влево (б)

Для циклического сдвига вправо (рис.4.8, а) нужно сохранить в дополнительной переменной последний элемент и на одну позицию вправо переместить подмассив, который начинается с нулевого элемента и заканчивается предпоследним. После этого на место нулевого элемента записывается элемент, который сохранен в дополнительной переменной и раньше был последним. Фрагмент кода, осуществляющий циклический сдвиг массива `a[]` на одну позицию вправо представлен ниже:

```
temp=a[a.length-1];
for(int i=a.length-1;i>0;i--)
    a[i]=a[i-1];
a[0]=temp;
```

При циклическом сдвиге влево (рис.4.8, б) в дополнительной переменной сохраняют нулевой элемент массива. Затем сдвигают на одну позицию влево подмассив, который начинается с первого элемента и заканчивается последним. На место последнего элемента записывают элемент, сохраненный в дополнительной переменной. Пример фрагмента кода, осуществляющий циклический сдвиг массива `a[]` на одну позицию влево представлен ниже:

```
temp=a[0];
for(int i=0;i<a.length-1;i++)
    a[i]=a[i+1];
a[a.length-1]=temp;
```

4.2.8 Проверка соответствия элементов массива некоторому условию

Алгоритмы проверок наличия в массиве элемента, удовлетворяющего условию, или соответствия всех элементов массива некоторому условию основываются на использовании дополнительной логической переменной, называемой *переменной-флаг*, которая изначально принимает значения *ложь*, а в случае выполнения проверяемого условия в цикле, принимает значение *истина*. Значения, которые следует присваивать переменной-флаг при организации различных проверок, представлены в табл. 4.5.

Таблица 4.5 – Значения переменной-флаг для разных типов проверок в одномерном массиве

<i>Проверка</i>	<i>Начальное значение</i>	<i>В цикле</i>
Наличие в массиве элемента, удовлетворяющего условию	flag=false	Если условие выполняется, то flag=true;
Отсутствие в массиве элемента, удовлетворяющего условию	flag=true	Если условие выполняется, то flag=false;
Все элементы массива удовлетворяют условию	flag=true	Если условие не выполняется, то flag=false;
Не все элементы массива удовлетворяют условию	flag=false	Если условие не выполняется, то flag=true.

Во всех четырёх случаях критичным является определение условия, при котором один элемент влияет на всю ситуацию в целом. В первых двух случаях таким условием является наличие элемента, удовлетворяющего условию, а в последних двух случаях – наличие элемента, не удовлетворяющего условию. Наличие одного элемента, влияющего на ситуацию в целом, позволяет организовывать досрочный выход из цикла с помощью оператора `break` при нахождении такого элемента.

Рассмотрим алгоритм проверки совпадения всех элементов массива. Задача может быть решена двумя разными способами, основывающимися на том факте, что если все элементы одинаковые, то они обязательно будут совпадать с нулевым элементом массива, это и есть проверяемое условие.

Первый способ сводится к проверке «все элементы массива удовлетворяют условию» (см. табл. 4.6) и предлагает определить, если ли в массиве хотя бы один элемент, отличающийся от нулевого. Для этого последовательно анализируются все элементы массива. Если какой-нибудь элемент отличается от нулевого, то делается вывод, что в массиве есть различные элементы и цикл прерывается. Фрагмент кода, иллюстрирующий организацию подобной проверки для массива `a[]` приведен ниже:

```
boolean flag = true;
for(int i=1; i<a.length;i++)
if(a[i]!=a[0]) {flag=false; break;}
System.out.printf("Элементы массива совпадают?-%b", flag);
```

Второй способ предлагает определить количество элементов, совпадающих с нулевым. Если это количество совпадет с общим количеством элементов в массиве, значит, массив состоит из одинаковых элементов:

```
int counter = 1;
for(int i=1; i<a.length;i++)
if(a[i]==a[0]) counter++;
System.out.printf("Элементы массива совпадают?-%b",
counter==a.length);
```

Еще одна интересная задача, требующая применения рассмотренных алгоритмов – проверка упорядоченности всех элементов массива.

Массив называется *упорядоченным (отсортированным) по возрастанию*, если каждый элемент массива не меньше всех предшествующих элементов и не больше всех последующих элементов. Другим словами, для любых двух элементов массива, отсортированного по возрастанию, выполняется условие:

Если $i < j$, то $a[i] \leq a[j]$,
где i и j – номера элементов массива,
 $a[i]$ и $a[j]$ – элементы массива с соответствующими номерами.

Массив называется *упорядоченным (отсортированным) по убыванию*, если каждый элемент массива не больше всех предшествующих элементов и не меньше всех последующих элементов. Другим словами, для любых двух элементов массива, отсортированного по убыванию, выполняется условие:

Если $i < j$, то $a[i] \geq a[j]$,
где i и j – номера элементов массива,
 $a[i]$ и $a[j]$ – элементы массива с соответствующими номерами.

Для проверки упорядоченности массива необходимо последовательно сравнить все соседние элементы массива. Если в массиве найдется хотя бы одна пара элементов, для которой не выполняется условие упорядоченности, значит, массив не отсортирован.

Рассмотрим особенности программной реализации данного алгоритма. Требуется проверить, упорядочены ли элементы целочисленного массива по возрастанию.

Задача 4.9. Проверьте, упорядочены ли элементы целочисленного массива по возрастанию.

Объяснение: в программе объявлена логическая переменная `flag`. До начала проверки ей присвоено значение `true`. В цикле попарно сравниваются элементы с номерами i и $(i+1)$. Чтобы при этом не выйти за границы массива, перебираются индексы с нулевого до предпоследнего. В цикле проверяется не выполнение условия упорядоченности по возрастанию: $a[i] > a[j]$. Если на какой-то из итераций цикла оно выполнится, переменной `flag` будет присвоено значение `false` и цикл будет прерван.


```

import java.util.Scanner;
public class Ex_4_9 {
    public static void main (String[] args){
        Scanner scan = new Scanner(System.in);
        System.out.println("Введите n:");
        int n=scan.nextInt();
        int [] a = new int[n];

        //Ввод элементов массива
        for (int i=0;i<a.length;i++){
            System.out.printf("Введите элемент a[%d]:", i+1);
            a[i]=scan.nextInt();}

        System.out.println("Исходный массив:");
        for (int x : a) System.out.printf("%d  ", x);

        boolean flag=true;
        for(int i=0; i<a.length-1;i++)
            if(a[i]>a[i+1]) {flag=false; break;}
        System.out.printf("%nЭлементы упорядочены по возрастанию?-%b",
        flag);
    }}

```

Результат:

```

Введите n:
5
Введите элемент a[1]:5
Введите элемент a[2]:6
Введите элемент a[3]:7
Введите элемент a[4]:8
Введите элемент a[5]:9
Исходный массив:
5 6 7 8 9
Элементы упорядочены по возрастанию?-true

```

4.3 Сортировка массива

Сортировка массива – это изменение порядка расположения его элементов таким образом, чтобы для каждой пары выполнялось заданное условие упорядоченности. Ранее уже были рассмотрены условия упорядоченности массивов по возрастанию и убыванию.

Существует множество различных методов сортировки массива, различающихся временем выполнения и объемом используемой памяти. В данном разделе рассмотрим наиболее известные элементарные (прямые) методы сортировки массива:

- сортировка простым обменом (метод пузырька);
- сортировка выбором;
- сортировка вставкой (метод прямого включения).

4.3.1 Сортировка методом пузырька

Сортировка методом пузырька основана на выполнении в цикле операций сравнения и при необходимости обмена соседних элементов. Процесс обмена продолжается до тех пор, пока все элементы не окажутся на своих местах, при которых выполняется условие упорядоченности. Рассмотрим алгоритм пузырьковой сортировки более подробно на примере.

Пусть задан массив, элементы которого необходимо отсортировать по возрастанию:

5	8	3	2	4
---	---	---	---	---

Расположим массив сверху вниз, от нулевого элемента — к последнему:

5
8
3
2
4

Организуем проход по элементам массива снизу вверх, по пути просматривая пары соседних элементов. Если элементы некоторой пары находятся в неправильном порядке, то меняем их местами (рис.4.9).

После нулевого шага ($i=0$) "вверх" оказывается самый "легкий" элемент — отсюда аналогия с пузырьком. Следующий шаг выполняется до второго сверху элемента, таким образом, второй по величине элемент поднимается на правильную позицию.

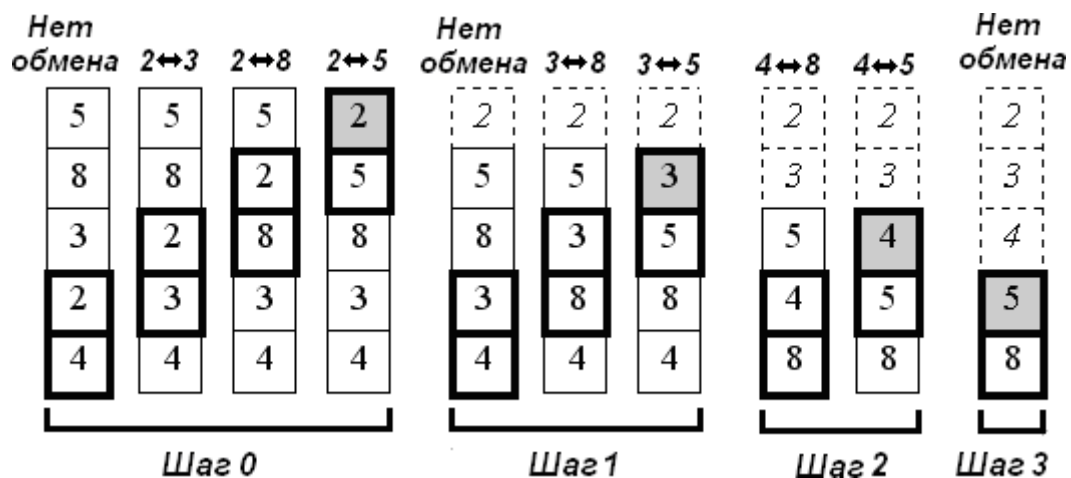


Рис.4.9 – Сортировка массива методом пузырька

Каждый следующий шаг выполняет проходы по все уменьшающейся нижней части массива до тех пор, пока в ней не останется только один элемент. На этом сортировка заканчивается, так как последовательность упорядочена по возрастанию. Блок-схема алгоритма сортировки массива по возрастанию методом пузырька представлена на рис.4.10. Для сортировки по убыванию надо поменять в блок-схеме знак «>» на знак «<».

Задача 4.10. Выполните сортировку целочисленного массива по возрастанию методом пузырька.

Объяснение: в программной реализации метода используются два вложенных цикла. Внешний цикл повторяется столько раз, сколько элементов необходимо поставить на нужное место, то есть $n-1$. В примере, массив состоял из пяти элементов ($n=5$), соответственно было выполнено $n-1=4$ шага.

Проход по элементам массива выполняет внутренний цикл. За каждый проход сравниваются два соседних элемента, и если порядок неверный элементы меняются местами с помощью дополнительной переменной *temp*. В каждом шаге проходы начинаются с последнего элемента и доходят до элемента индекс, которого соответствует номеру выполняемого шага.

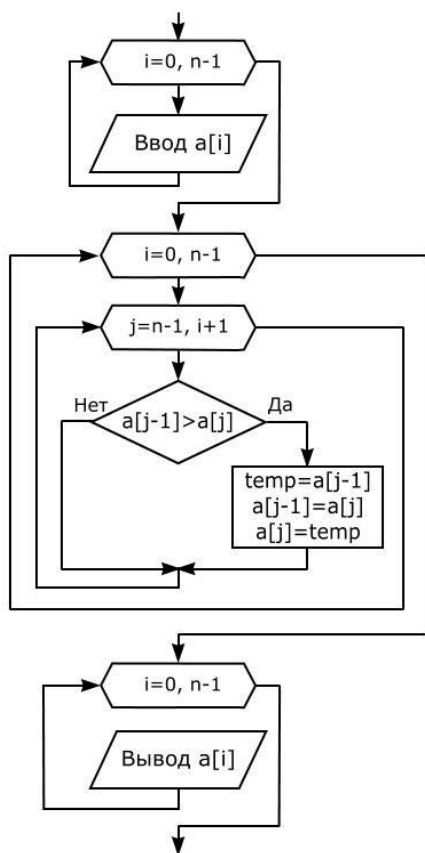


Рис.4.10 – Блок-схема алгоритма сортировки методом пузырька

```

import java.util.Scanner;
public class Ex_4_10 {
    public static void main (String[] args){
        Scanner scan = new Scanner(System.in);

        System.out.println("Введите n:");
        int n=scan.nextInt();
        int [] a = new int[n];
        //заполнение случайными числами
        for(int i=0;i<a.length;i++)
            a[i]=((int) (Math.random()*90)+9);
        System.out.println("Исходный массив:");
        for (int x : a) System.out.printf("%d ", x);

        // сортировка пузырьком по возрастанию
        int temp;
        for(int i=0; i<a.length; i++) { // i - номер шага
            for(int j=a.length-1; j>i; j-- ) {
                if ( a[j-1] > a[j] ) {
                    temp=a[j-1];
                    a[j-1]=a[j];
                    a[j]=temp;
                }
            }
        }
        System.out.printf("%nОтсортированный массив:%n");
        for (int x : a) System.out.printf("%d ", x);
    }
}

```

Результат:

```

Введите n:
10
Исходный массив:
12 82 63 22 57 36 25 63 79 67
Отсортированный массив:
12 22 25 36 57 63 63 67 79 82

```

4.3.2 Сортировка методом выбора

На каждом шаге выполнения метода массив разделен на две части: левую, уже отсортированную и правую, еще не отсортированную. Если выполняется сортировка по возрастанию, то первоначально в массиве отыскивается минимальный элемент и обменивается местами с нулевым элементом. Затем, из оставшихся неотсортированных элементов, опять выбирается минимальный элемент и обменивается с первым из неотсортированных элементов. В результате отсортированная часть массива увеличивается на один элемент, а неотсортированная – на один элемент уменьшается. Процесс продолжается до тех пор, пока в неотсортированной части останется один элемент.

Рассмотрим алгоритм выполнения метода выбора на примере. Пусть задан массив, элементы которого необходимо отсортировать по возрастанию:

5	8	3	2	4
---	---	---	---	---

Последовательность выполняемых шагов сортировки массива представлена на рис.4.11. Пунктирной линией выделена отсортированная часть массива, сплошной – еще неотсортированная часть. Вне зависимости от номера текущего шага i , последовательность $a[0] \dots a[i]$ (выделенная курсивом) является упорядоченной. Таким образом, на $(n-1)$ -м шаге вся последовательность, кроме последнего элемента оказывается отсортированной, все меньшие элементы уже ушли влево и это значит, что массив упорядочен по возрастанию.

Блок-схема алгоритма сортировки массива по возрастанию методом выбора представлена на рис.4.12. Для сортировки по убыванию надо поменять в блок-схеме знак «>» на знак «<».



Рис.4.11 – Сортировка массива методом выбора

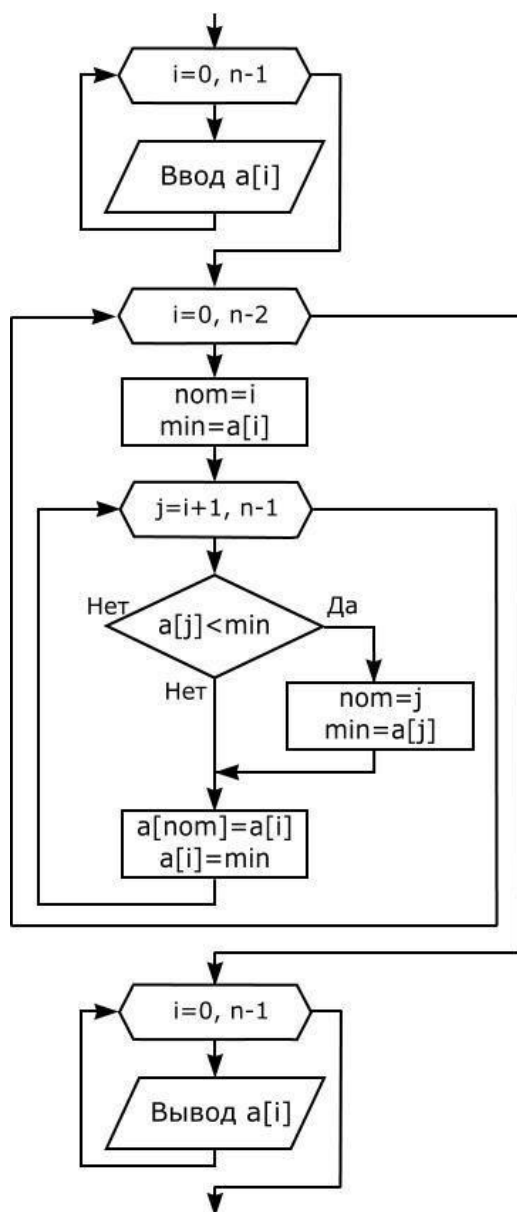


Рис.4.12 – Блок-схема алгоритма сортировки методом выбора

Задача 4.11. Выполните сортировку целочисленного массива по возрастанию методом выбора.

Объяснение: в программе два вложенных цикла. Внешний цикл повторяется $n-1$ раз. В начале каждой итерации внешнего цикла параметр i – индекс первого неотсортированного элемента в правой части массива, с которым будет проведен обмен.

Во внутреннем цикле осуществляется поиск минимального элемента и его номера в неотсортированной части, т.е. от элемента с индексом $i+1$ и до конца массива. Минимальный элемент сохраняется в переменную

`min`, которая должна иметь такой же тип, что и элементы массива. Номер найденного элемента запоминается в целой переменной `nom`.

После того, как минимальный элемент найден, он обменивается с первым неотсортированным элементом, имеющим индекс `i`. После чего элемент с индексом `i` замыкает отсортированную левую часть массива, увеличивая ее длину на 1.

```
import java.util.Scanner;
public class Ex_4_11 {
    public static void main (String[] args){
        Scanner scan = new Scanner(System.in);

        System.out.println("Введите n:");
        int n=scan.nextInt();
        int [] a = new int[n];

        //заполнение случайными числами
        for(int i=0;i<a.length;i++)
            a[i]=((int) (Math.random()*90)+9);

        System.out.println("Исходный массив:");
        for (int x : a) System.out.printf("%d  ", x);

        // сортировка методом выбора по возрастанию
        int nom;
        int min;
        for(int i=0;i<a.length-1;i++) { //i-номер текущего шага
            nom=i; min=a[i];
            for(int j=i+1; j<a.length; j++)
                // находим наименьший элемент и его номер
                if (a[j]< min) {
                    nom=j; min=a[j];
                }
            a[nom] = a[i]; a[i] = min; //обмен
        }
        System.out.printf("%nОтсортированный массив:%n");
        for (int x : a) System.out.printf("%d  ", x);
    }
}
```

Результат:

```
Введите n:
10
Исходный массив:
36  58  77  36  83  83  30  17  56  67
Отсортированный массив:
17  30  36  36  56  58  67  77  83  83
```

4.3.3 Сортировка методом вставки

В сортировке вставками, как и в вышеизложенных методах, выполняются проходы по части массива, и массив разделен на две части, левую упорядоченную и правую, еще не упорядоченную. Однако в сортировке пузырьком или выбором можно было четко заявить, что на i -м шаге элементы $a[0] \dots a[i]$ стоят на правильных местах и никуда более не переместятся. В данном же методе предполагается, что по ходу выполнения алгоритма сортировки в отсортированную часть будут вставляться все новые элементы.

Разберем алгоритм на примере сортировки массива по возрастанию. Задан исходный одномерный массив вида:

5	8	3	2	4
---	---	---	---	---

Для наглядности расположим элементы массива вертикально (рис. 4.13). Рассмотрим действия, выполняемые в соответствии с алгоритмом, на i -м шаге. Последовательность к этому моменту разделена на две части: упорядоченную $a[0] \dots a[i]$ и неупорядоченную $a[i+1] \dots a[n-1]$.

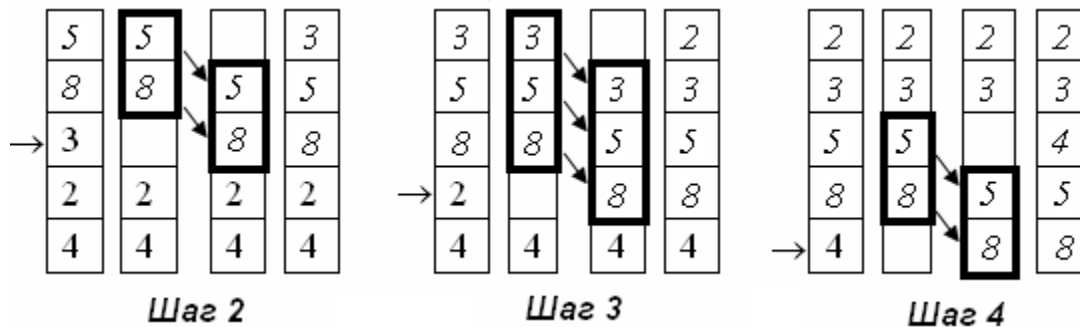


Рис.4.13 – Сортировка массива методом вставки

На следующем, $(i+1)$ -м каждом шаге алгоритма элемент $a[i+1]$ сохраняется в дополнительной переменной. Далее проводится поиск подходящего для него места в упорядоченной части массива путем последовательных сравнений с элементом, стоящим перед ним. В зависимости от результата сравнения элемент либо остается на текущем месте (вставка завершена), либо сдвигается на одну позицию вправо, освобождая место для вставки, и процесс повторяется до тех пор, пока не будет:

- 1) найден элемент, меньший вставляемого элемента;
- 2) достигнуто начало последовательности.

Блок-схема алгоритма сортировки массива по возрастанию методом вставки представлена на рис.4.14. Для сортировки по убыванию надо поменять в блок-схеме знак «>» на знак «<».

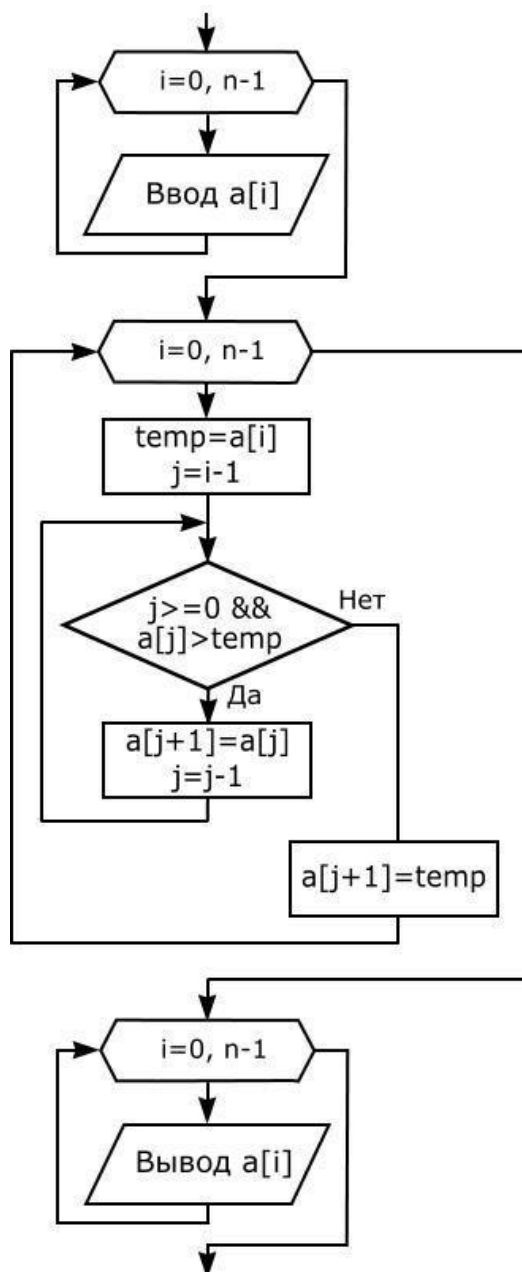


Рис.4.14 – Блок-схема алгоритма сортировки методом вставки

Задача 4.12. Выполните сортировку целочисленного массива по возрастанию методом вставки.

Объяснение: в программе два вложенных цикла. Внешний цикл повторяется n раз, так как алгоритм повторяется для каждого элемента массива начиная с нулевого. Параметр i – индекс элемента, который нужно вставить в отсортированную часть массива. В начале выполнения цикла значение этого элемента $a[i]$ сохраняется в дополнительную переменную $temp$. Далее значение этого элемента сравнивается с элементами стоящими перед ним, начиная с элемента с индексом $i-1$.

Сравнения проводят во внутреннем цикле и продолжаются до тех пор, пока впереди стоящий элемент $a[j]$ больше вставляемого элемента $a[j] > temp$ или пока не будет достигнуто начало массива $j \geq 0$. Во внутреннем цикле элементы отсортированной части массива при выполнении перечисленных условий сдвигаются вправо, освобождая нужное место для вставки.

```
import java.util.Scanner;
public class Ex_4_12 {
    public static void main (String[] args){
        Scanner scan = new Scanner(System.in);

        System.out.println("Введите n:");
        int n=scan.nextInt();
        int [] a = new int[n];

        //заполнение случайными числами
        for(int i=0;i<a.length;i++)
            a[i]=((int) (Math.random()*90)+9);

        System.out.println("Исходный массив:");
        for (int x : a) System.out.printf("%d ", x);

        // сортировка методом вставки по возрастанию
        int i, j, temp;
        for (i=0; i<a.length; i++) {    //i - номер шага
            temp = a[i];
            j=i-1;
            while(j>=0 && a[j]>temp){
                // сдвигаем элементы вправо, пока не найдем
                // место для вставки
                a[j+1] = a[j];
                j--;
            }
            a[j+1] = temp; //вставляем элемент
        }
        System.out.printf("\nОтсортированный массив:\n");
        for (int x : a) System.out.printf("%d ", x);
    }
}
```

Результат:

```
Введите n:
10
Исходный массив:
14  94  59  16  15  32  57  85  87  79
Отсортированный массив:
14  15  16  32  57  59  79  85  87  94
```

Алгоритмы сортировок методами выбора, вставки и пузырька не требуют дополнительной памяти. Время выполнения выше изложенных

алгоритмов пропорционально количеству операций сравнения или перестановок элементов. В пузырьковой сортировке количество сравнений всегда одно и то же, поскольку два цикла `for` повторяются указанное количество раз независимо от того, был список изначально упорядочен или нет. Это значит, что алгоритм пузырьковой сортировки всегда выполняет $(n^2 - n)/2$ сравнений, где n – количество сортируемых элементов. Данная формула выведена на том основании, что внешний цикл выполняется $n - 1$ раз, а внутренний выполняется в среднем $n / 2$ раз.

В сортировке методом выбора, как и в пузырьковой сортировке, внешний цикл выполняется $n - 1$ раз, а внутренний – в среднем $n / 2$ раз. Следовательно, сортировка методом выбора требует $(n^2 - n)/2$ сравнений. Таким образом, это алгоритм порядка n^2 , из-за чего он считается слишком медленным для сортировки большого количества элементов. Несмотря на то, что количество сравнений у двух методов одинаковое, в сортировке выбором количество обменов в среднем случае намного меньше, чем в пузырьковой сортировке.

Количество сравнений в сортировке вставками зависит от изначальной упорядоченности массива. Если массив уже отсортирован, количество сравнений равно $n - 1$, иначе его производительность является величиной порядка n^2 . Отметим, что применение простых сортировок эффективно на небольших объемах данных. В противном случае, лучшим вариантом является применение усовершенствованных методов сортировки.

4.4 Работа с последовательностями

В отдельных случаях при решении задач, связанных с обработкой последовательно поступающих элементов, для экономии памяти и времени можно не запоминать их в массиве. Это особенно актуально при вычислении разных характеристик элементов последовательности, а именно: суммы, произведения, количества, максимальных и минимальных элементов и др. Определяются подобные характеристики для последовательностей аналогично тому, как они определялись для одномерных массивов. Однако отсутствие массивов накладывает определенные ограничения на организацию программы.

Так, следует учесть то, что может быть введена сколь угодно длинная последовательность и количество элементов в ней заранее не известно. Поэтому ввод данных осуществляют в бесконечном цикле. Прерывают цикл либо по вводу определенного значения, заранее установленному как признак окончания ввода, либо в случае, возникновения *исключительной ситуации*. Исключительная ситуация может возникнуть при неверном

вводе очередного числа (например, был введен символ). Для обработки исключительной ситуации можно использовать специальную конструкцию `try-catch`. Она имеет два блока. В блок `try` помещают фрагмент кода, при выполнении которого возможно возникновение исключительной ситуации. В блок `catch` – код, который должен быть выполнен, если исключительная ситуация возникнет. Интерпретатор байт-кода попытается выполнить все действия, указанные в блоке `try`. Если это удастся, то содержимое блока `catch` не оказывает никакого влияния на ход выполнения программы. Однако если в процессе выполнения блока `try` возникает исключительная ситуация, то управление немедленно передается блоку `catch`, а затем программа завершает работу.

При работе с последовательностями, возникают дополнительные сложности с установкой начальных значений целевых переменных. Так, например, при поиске максимального или минимального элемента одномерного массива в соответствии с алгоритмом в качестве начального значения принимается нулевой элемент массива. Однако ввод и обработка элементов последовательности происходит в бесконечном цикле, поэтому заранее не известно значение первого введенного элемента. В этом случае в качестве начального значения для максимума или минимума выбирают наименьшее (наибольшее) значение из диапазона допустимых значений базового типа элементов последовательности. Для этого в Java для каждого примитивного типа определены специальные классы-оболочки, в которых имеются соответствующие константы – `MAX_VALUE` и `MIN_VALUE`. Список констант приведен в табл. 4.6.

Обратите внимание на то, что константа `MIN_VALUE` для типа `double` и `float` возвращает минимальное *положительное* значение диапазона. Чтобы получить действительно минимальные значения диапазонов данных вещественных типов используйте выражения: `-1*Double.MAX_VALUE` и `-1*Float.MAX_VALUE`.

Таблица 4.6 – Константы `MAX_VALUE` и `MIN_VALUE` для примитивных типов

<i>Примитивный тип</i>	<i>Константы</i>	
	<i>минимальное значение диапазона</i>	<i>максимальное значение диапазона</i>
byte	Byte.MIN_VALUE	Byte.MAX_VALUE
short	Short.MIN_VALUE	Short.MAX_VALUE
int	Integer.MIN_VALUE	Integer.MAX_VALUE

long	Long.MIN_VALUE	Long.MAX_VALUE
char	Character.MIN_VALUE	Character.MAX_VALUE
	<i>минимальное положительное значение диапазона</i>	<i>максимальное значение диапазона</i>
float	Float.MIN_VALUE	Float.MAX_VALUE
double	Double.MIN_VALUE	Double.MAX_VALUE

Задача 4.13. Напишите программу, вводящую последовательность целых чисел, и печатающую ее максимальный элемент.

Объяснение: в блок try помещен бесконечный цикл, что гарантирует выход из него только при возникновении исключительной ситуации – ошибки при вводе очередного числа. Если это случается, то управление передается в блок catch, в котором находится оператор вывода результата.

```
import java.util.Scanner;
public class Ex_4_13 {
    public static void main(String[] args) {
        Scanner scan = new Scanner (System.in);
        int x, max = Integer.MIN_VALUE;
        try {
            while (true) {
                System.out.printf("Введите элемент:");
                x = scan.nextInt();
                if (x > max) max = x; }
        } catch (Exception e) {System.out.printf("Max=%d", max);}
    }}

```

Результат:

```
Введите элемент:5
Введите элемент:8
Введите элемент:6
Введите элемент:9
Введите элемент:7
Введите элемент:2
Введите элемент:1
Введите элемент:3
Введите элемент:7
Введите элемент:h
Max=9

```

4.5 Многомерные массивы

Как было изложено выше, одномерные массивы используются для хранения последовательностей данных одного типа. Базовым типом

массива может быть любой тип Java, т.е. массив может содержать элементы не только примитивных типов, но и объекты, в том числе и другие массивы. Массив, который в качестве своих элементов содержит другие массивы, называется *многомерным массивом*.

В программировании наиболее часто используются двумерные массивы (матрицы). Каждая строка матрицы является обычным одномерным массивом, а объединение всех строк – двумерным массивом, в каждом элементе которого хранится ссылка на какую-то строку матрицы. Иными словами, двумерный массив представляет собой массив ссылок на одномерные массивы, трехмерный массив представляет собой список ссылок, но уже на двумерные массивы, а n-мерный массив – это одномерный массив, элементами которого являются ссылки на массивы размерности n-1. Рассмотрим более подробно особенности описания многомерных массивов в Java и базовые алгоритмы обработки двумерных массивов.

4.5.1 Описание многомерных массивов

При объявлении переменной многомерного массива для указания каждого дополнительного индекса используют отдельный набор квадратных скобок. Рассмотрим примеры объявления ссылок на массивы разной размерности:

```
double[] a1;           // одномерный массив
double[][] a2;         // двумерный массив
double[][][] a3;       // трехмерный массив
double[][][][][] a5;   // пятимерный массив
```

Альтернативной формой объявления имени массива является следующая запись:

```
double a2[][][];
```

Существует несколько способов определения и инициализации многомерных массивов.

При определении многомерного массива можно явно указать размерность каждого индекса:

```
a2 = double[3][5];
```

В данном случае массив представляет собой прямоугольную матрицу, состоящую из 3 строк и 5 столбцов (рис.4.15). Правильнее было бы сказать, что определен одномерный массив, состоящий из 3 элементов, где каждый из этих элементов также является одномерным массивом, состоящим из 5 элементов (*массив массивов*).

		Правый индекс (столбец)				
		0	1	2	3	4
Левый индекс (строка)	0	a2[0][0]	a2[0][1]	a2[0][2]	a2[0][3]	a2[0][4]
	1	a2[1][0]	a2[1][1]	a2[1][2]	a2[1][3]	a2[1][4]
	2	a2[2][0]	a2[2][1]	a2[2][2]	a2[2][3]	a2[2][4]

Рис.4.15 – Двумерный массив размерностью 3x5

Для обращения к элементам многомерных массивов, как и в случае одномерных массивов, используется операция индексирования. Для двумерных массивов указывается индекс строки и индекс столбца. Индексация начинается с нуля. Например, a2[2][1] – элемент двумерного массива, расположенный в третьей строке и во втором столбце.

Описание трехмерного массива можно выполнить следующим образом:

```
int b[][][] = new int [3][4][5];
```

Будет создан массив массивов, состоящий из 3 элементов, представляющих собой список ссылок на двумерные массивы размерностью 4x5 элементов, также являющиеся массивами массивов.

При определении многомерного массива можно явно указать только размерность по первому индексу:

```
int [][] a = new int[3][];
```

Такое определение позволяет задать для массива произвольную форму, указав разную длину строк:

```
a[0] = new int[3];
a[1] = new int[1];
a[2] = new int[2];
```

В результате будет создан массив, содержащий три массива, первый из которых имеет длину три элемента, второй один элемент, а третий два элемента (рис.4.16). Такой массив уже не является матрицей, так как в каждой его строке разное количество элементов. Следует отметить, что чаще используются двумерные массивы с равным количеством элементов в каждой строке.

Обратите внимание, что при таком объявлении длина массива по первому индексу равна 3. Ее можно получить через поле a.length. Элементами этого массива являются ссылки на массивы с данными. Длина каждого из этих массивов может быть получена так же через поле length,

которое уточняется индексом элемента. Так, `a[0].length` – длина нулевого элемента, `a[1].length` – первого и т.д.

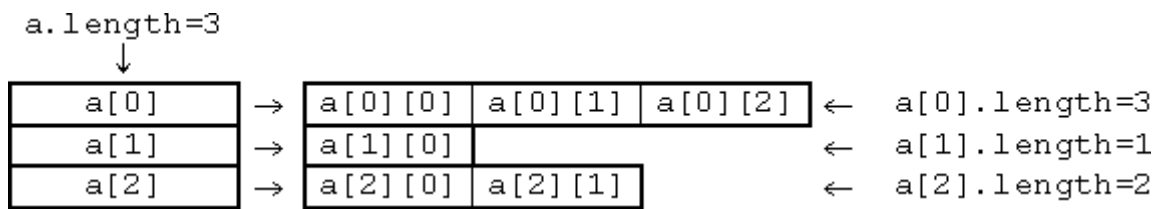


Рис.4.16 – Двумерный массив произвольной формы

Как и в случае с одномерными массивами, многомерные массивы можно объявлять, определять и инициализировать в одном операторе, заключив список инициализации в фигурные скобки, например:

```
int b=2;
int [][] a = {{2,5,6,7}, null, {}, {b, 2*b,1}};
```

Причем внутри инициализаторов массивов можно применять как литеральное значение, так и выражения. В примере создан массив массивов длиной 4, и три массива с длинами 4, 0 и 3 соответственно.

4.5.2 Базовые алгоритмы обработки двумерных массивов

Для обработки элементов двумерных массивов удобно использовать два вложенных друг в друга цикла. Параметр-счетчик одного цикла осуществляет обход по строкам матрицы, счетчик другого цикла – по столбцам. Таким образом, сначала обрабатываются все элементы нулевой строки (столбца), потом – все элементы первой строки (столбца) и т.д.

Рассмотрим пример заполнения элементов двумерного массива случайными числами и вывод его на экран в виде матрицы.

```
int n=10, m=10; //количество строк и столбцов массива
int[][] a = new int[n][m];
//заполнение случайными числами
for(int i=0; i<a.length; i++) //обход по строкам
    for(int j=0; j<a[i].length; j++) //обход по столбцам
        a[i][j] = (int)(Math.random()*10);

//вывод массива на экран
for(int i=0; i<a.length; i++) {
    for(int j=0; j<a[i].length; j++)
        System.out.printf("%5d",a[i][j]);
    //Переход на следующую строку
    System.out.println();
}
```


Обратите внимание, в примере для того, чтобы обеспечить вывод матрицы на экран в виде таблицы, после завершения вывода всех элементов текущей строки во внешнем цикле выполняется перевод курсора на новую строку.

Программный код, осуществляющий ввод элементов массива с клавиатуры представлен ниже:

```
Scanner scan = new Scanner(System.in);
for(int i=0; i<a.length; i++)
    for(int j=0; j<a[i].length; j++){
        System.out.printf("Введите a[%d][%d]->%n", i, j);
        a[i][j] = scan.nextInt();
    }
```

Для двумерных массивов, как и для одномерных массивов, применима усовершенствованная версия цикла `for`. При организации вложенных циклов итерационная переменная внешнего цикла должна быть ссылкой на одномерный массив. Рассмотрим пример вывода на экран двумерного массива с использованием версии `for-each` цикла `for`.

```
double[][] a = new double[n][m];
//вывод массива на экран
for(double x[] : a) {
    for(double y : x)
        System.out.printf("%5.2f", y);
        //Переход на следующую строку
    System.out.println();
}
```

Обратите внимание, переменная `x` объявлена как ссылка на одномерный массив вещественных значений. Это необходимо, потому что результат выполнения каждой итерации цикла `for` – следующий массив в массиве `a`, начиная с массива, указанного элементом `a[0]`. Затем внутренний цикл `for` перебирает каждый из этих массивов, отображая значения каждого элемента.

Задача 4.13. Заполните квадратную матрицу размерностью 10×10 элементов по заданному образцу.

1	0	0	...	0	0	1
0	2	0	...	0	2	0
0	0	3	...	3	0	0
:	:	:	...	:	:	:
0	0	8	...	8	0	0
0	9	0	...	0	9	0
10	0	0	...	0	0	10

Объяснение: перед решением задачи целесообразно вспомнить некоторые свойства матриц (рис.4.17):

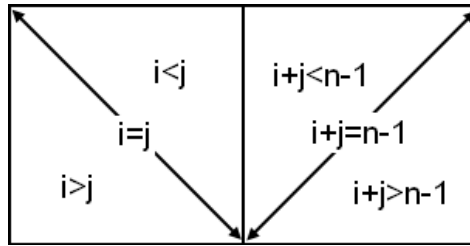


Рис. 4.17 – Свойства элементов матрицы

– диагональ, которая проходит через верхний левый и нижний правый углы квадратной матрицы называется *главной диагональю*. У элементов, расположенных на главной диагонали совпадает индекс строки и столбца ($i=j$);

– если номер строки превышает номер столбца ($i>j$), то элемент находится ниже главной диагонали;

– если номер строки меньше номера столбца ($i<j$), то элемент находится выше главной диагонали;

– диагональ, которая проходит через верхний правый и нижний левый углы квадратной матрицы называется *побочной диагональю*. У элементов, расположенных на побочной диагонали сумма индексов равна $n-1$, где $n \times n$ – размерность матрицы ($i+j==n-1$).

– неравенство $i+j < n-1$ характерно для элемента находящегося выше побочной диагонали;

– соответственно, элементу, лежащему ниже побочной диагонали, соответствует выражение $i+j > n-1$.

```
public class Ex_4_13 {
    public static void main (String[] args){
        int n=10;
        int [][] a = new int[n][n];

        //заполнение матрицы
        for(int i=0;i<a.length;i++)
            for(int j=0;j<a[i].length;j++)
                if (i==j||i+j==n-1) a[i][j]=i+1;

        for (int x[] : a){
            for (int y : x)
                System.out.printf("%3d", y);
            System.out.println();}
    }}

```

Рассмотрим другие базовые алгоритмы обработки двумерных массивов на примере некоторых типовых задач. Их решение ничем

принципиально не отличается от решения аналогичных задач для одномерных массивов. Нахождение суммы, количества, произведения, максимальных и минимальных элементов для матрицы выполняется в соответствии с рассмотренными ранее алгоритмами.

Отметим, что при работе с двумерными массивами можно находить перечисленные характеристики не только для всей матрицы, но и по строкам и по столбцам отдельно. В этом случае, результатом будет не одна целевая переменная, а одномерный массив значений, размерность которого соответствует количеству строк, если характеристика рассчитывалась по строкам, или столбцов – в противном случае.

Ниже приведены фрагменты программ, вычисляющие некоторые характеристики для целочисленной матрицы $a[n][m]$.

Количество положительных элементов матрицы:

```
int counter=0;
for(int i=0;i<a.length;i++)
    for(int j=0;j<a[i].length;j++)
        if (a[i][j]>0) counter++;
```

И этот же фрагмент, но с применением усовершенствованного цикла for:

```
int counter=0;
for (int x[] : a)
    for (int y : x)
        if (y>0) counter++;
```

Произведение элементов матрицы, расположенных выше главной диагонали (при условии, что $n=m$):

```
int pr=1;
for(int i=0;i<a.length;i++)
    for(int j=0;j<a[i].length;j++)
        if (i<j) pr*=a[i][j];
```

Условием расположения элементов матрицы выше главной диагонали является неравенство $i < j$, где i – номер строки, j – номер столбца, на пересечении которых данный элемент находится.

Максимальный элемент матрицы, а также индексы строки и столбца, на пересечении которых он расположен:

```
int max=a[0][0], nomi=0, nomj=0;
for(int i=0;i<a.length;i++)
    for(int j=0;j<a[i].length;j++)
        if (a[i][j]>max) {max=a[i][j]; nomi=i; nomj=j;}
```

Обратите внимание, что в случае двумерного массива при поиске максимального элемента (как и минимального) индексы строк и столбцов следует перебирать, начиная с нуля. В примере $nomi$ – индекс строки, $nomj$ – индекс столбца, в которых расположен максимальный элемент max .

Количество минимальных элементов по строкам матрицы:

```
int[] counter = new int[n]; //массив для количеств
for(int i=0; i<a.length; i++){
    int min=a[i][0]; counter[i]=0;
    for(int j=0; j<a[i].length; j++){
        if (a[i][j]<min) {min=a[i][j]; counter[i]=1;}
        else if (a[i][j]==min) counter[i]++; }
    }
```

В примере найденные количества минимальных элементов по строкам сохраняются в массив counter[], который имеет длину равную количеству строк матрицы.

Среднее арифметическое значение элементов в каждом столбце матрицы:

```
double[] sr = new double [m];
for(int j=0; j<m; j++){
    sr[j]=0;
    for(int i=0; i<n; i++){
        sr[j]+=a[i][j];
    }
    sr[j]/=n; }
```

Средние арифметические значения элементов по столбцам сохраняются в массиве sr[], размерность которого равна количеству столбцов матрицы. Внешний цикл осуществляет обход по столбцам, а внутренний цикл перебирает и складывает элементы, расположенные в данном столбце.

В задачах на перестановку отдельных элементов массива, столбцов или строк используется алгоритм обмена значениями двух переменных через третью переменную (возможны и другие способы обмена значениями двух переменных).

Задача 4.14. Дан вещественный двумерный массив размерностью *n*х*m* элементов. Найдите сумму элементов в каждой строке матрицы. Строку с наибольшей и наименьшей суммой элементов поменяйте местами.

```
import java.util.Scanner;
public class Ex_4_14 {
    public static void main (String[] args){
        Scanner scan = new Scanner (System.in);
        System.out.printf("Введите n:");
        int n=scan.nextInt();
        System.out.printf("Введите m:");
        int m=scan.nextInt();
        double [][] a = new double[n][m];
```

```

        for(int i=0;i<a.length;i++)
            for(int j=0;j<a[i].length;j++)
                a[i][j]=Math.random()*10;

//вычисление суммы элементов по строкам
double[] sum = new double [n];
for(int i=0;i<a.length;i++){
    sum[i]=0;
    for(int j=0;j<a[i].length;j++)
        sum[i]+=a[i][j];}
System.out.printf("Исходная матрица%40s:%n",
                    "Сумма элементов");

    for(int i=0;i<a.length;i++){
        for(int j=0;j<a[i].length;j++)
            System.out.printf("%5.1f", a[i][j]);
        System.out.printf("%10.1f%n", sum[i]);
    }

//номера строк с максимальной и минимальной суммой
int nommax=0, nommin=0;
for(int i=1;i<n;i++) {
    if (sum[i]>sum[nommax]) nommax=i;
    if (sum[i]<sum[nommin]) nommin=i;
}
//обмен строк
double temp;
for(int j=0;j<m;j++)
{temp=a[nommax][j];
a[nommax][j]=a[nommin][j];
a[nommin][j]=temp;
}
System.out.printf("Новая матрица:%n");
for(int i=0;i<a.length;i++){
    for(int j=0;j<a[i].length;j++)
        System.out.printf("%5.1f", a[i][j]);
        System.out.println();
    }
}
}

```

Результат:

Введите n:5

Введите m:8

Исходная матрица

2,4	3,1	9,1	3,2	4,0	9,5	9,6	1,0	42,0
1,8	3,1	9,7	9,4	5,4	6,4	3,7	5,4	45,0
0,5	6,4	7,4	6,2	7,9	3,5	9,5	0,8	42,2
8,9	7,9	2,5	9,8	5,2	9,1	0,8	7,3	51,5
5,6	10,0	1,0	6,8	3,8	3,1	7,2	0,6	37,9

Сумма элементов:

Новая матрица:

2,4	3,1	9,1	3,2	4,0	9,5	9,6	1,0
1,8	3,1	9,7	9,4	5,4	6,4	3,7	5,4
0,5	6,4	7,4	6,2	7,9	3,5	9,5	0,8
5,6	10,0	1,0	6,8	3,8	3,1	7,2	0,6

8,9 7,9 2,5 9,8 5,2 9,1 0,8 7,3

Проверки соответствия элементов матрицы определенному условию удобно выполнять с использованием логической переменной-флаг. Как, например, это выполнено в задаче 4.15.

Задача 4.15. Проверьте, является ли квадратная матрица симметричной относительно главной диагонали.

```
import java.util.Scanner;
public class Ex_4_15 {
    public static void main (String[] args){
        Scanner scan = new Scanner (System.in);
        System.out.printf("Введите n:");
        int n=scan.nextInt();
        int [][] a = new int[n][n];

        for(int i=0;i<a.length;i++)
            for(int j=0;j<a[i].length;j++){
                System.out.printf("Введите a[%d][%d]->%n",i,j);
                a[i][j] = scan.nextInt();
            }

        System.out.printf("Исходная матрица:%n");
        for(int i=0;i<a.length;i++){
            for(int j=0;j<a[i].length;j++)
                System.out.printf("%3d", a[i][j]);
            System.out.println();
        }

        //проверка условия симметричности матрицы
        boolean flag=true;
        for(int i=0;i<a.length;i++)
            for(int j=0;j<a[i].length/2;j++)
                if(a[i][j]!=a[j][i]) flag=false;

        System.out.printf("Матрица симметричная - %b",flag);
    }
}
```

Результат:

```
Введите n:3
Введите a[0][0]->
1
Введите a[0][1]->
2
Введите a[0][2]->
3
Введите a[1][0]->
2
Введите a[1][1]->
1
Введите a[1][2]->
4
```

```
Введите a[2][0]->
3
Введите a[2][1]->
4
Введите a[2][2]->
1
Исходная матрица:
  1  2  3
  2  1  4
  3  4  1
Матрица симметричная - true
```

4.5.3 Сортировка двумерных массивов

Сортировка двумерных массивов сводится к упорядочиванию элементов, объединенных в столбцы или строки. Например, сортировка по убыванию элементов столбцов означает, что элементы следует расположить по убыванию сверху вниз в каждом столбце отдельно. При этом, рассматривая строку или столбец как одномерный массив, к ним применяют алгоритмы сортировок одномерных массивов.

Ниже приведен фрагмент программного кода, выполняющего сортировку строк двумерного целочисленного массива по возрастанию методом пузырька:

```
int temp;
for (int i=0; i<a.length; i++)//перебираем строки
    //сортируем элементы в строке методом пузырька
    for(int j=0; j<m; j++){
        for(int k=m-1; k>j; k-- ) {
            if ( a[i][k-1]>a[i][k] ) {
                temp=a[i][k-1];
                a[i][k-1]=a[i][k];
                a[i][k]=temp;}}}

```

Можно рассматривать строки и столбцы как отдельные элементы, которые следует упорядочить по заданному ключу. В этом случае в соответствии с выбранным алгоритмом сортировки, нужно обменивать не элементы, а сами строки или столбцы.

Задача 4.15. Упорядочить методом выбора строки матрицы по возрастанию максимальных элементов в строках.

```
import java.util.Scanner;
public class Ex_4_15 {
    public static void main (String[] args){
        Scanner scan = new Scanner (System.in);
        System.out.printf("Введите n:");
        int n=scan.nextInt();
    }
}
```

```

System.out.printf("Введите m:");
int m=scan.nextInt();
double [][] a = new double[n][m];

for(int i=0;i<a.length;i++)
for(int j=0;j<a[i].length;j++){
    a[i][j] = Math.random()*10-5;}

/*Нахождение максимальных элементов по строкам*/
double max[]=new double[a.length];
for(int i=0;i<a.length;i++)
{ max[i]=a[i][0];
  for(int j=1;j<a[i].length;j++)
    if(a[i][j]>max[i]) max[i]=a[i][j];}

System.out.printf("Исходная матрица%40s%n", "Максим. эл.");
for(int i=0;i<a.length;i++){
  for(int j=0;j<a[i].length;j++)
    System.out.printf("%5.1f", a[i][j]);
    System.out.printf("%10.1f%n",max[i]);
  }

  //Сортировка матрицы методом выбора
  double c[]=new double[m]; //вспомогательный массив,
                             // для промежуточного хранения

  int nom;
  double min;
  for( int i=0; i < a.length-1; i++) {
    nom=i; min=max[i];
    for(int j=i+1; j < a.length; j++)
      if (max[j] < min) {
        nom=j; min=max[j];    }
    max[nom] = max[i]; max[i] = min;
    c=a[nom]; a[nom]=a[i]; a[i]=c;}

  System.out.printf("Упорядоченная матрица%34s%n",
"Максим. эл.");
  for(int i=0;i<a.length;i++){
    for(int j=0;j<a[i].length;j++)
      System.out.printf("%5.1f", a[i][j]);
      System.out.printf("%10.1f%n", max[i]);
    }
  }
}

```

Результат:

Введите n:5

Введите m:8

Исходная матрица

3,4	2,6	1,2	-4,5	0,9	-2,3	-0,4	-0,1
-2,7	-1,3	2,7	4,5	2,0	1,6	3,1	0,5
-0,9	-0,8	-2,3	0,6	4,6	-0,8	-4,6	3,9
-4,1	-1,3	3,9	-0,8	-2,5	3,7	-4,8	3,3
-1,4	4,5	-4,2	3,7	3,1	4,2	-0,1	2,2

Максим. эл.

3,4
4,5
4,6
3,9
4,5

Упорядоченная матрица	Максим. эл.
3,4 2,6 1,2 -4,5 0,9 -2,3 -0,4 -0,1	3,4
-4,1 -1,3 3,9 -0,8 -2,5 3,7 -4,8 3,3	3,9
-1,4 4,5 -4,2 3,7 3,1 4,2 -0,1 2,2	4,5
-2,7 -1,3 2,7 4,5 2,0 1,6 3,1 0,5	4,5
-0,9 -0,8 -2,3 0,6 4,6 -0,8 -4,6 3,9	4,6

Упражнения к главе 4

1. Чему будет равна переменная `x` после следующего объявления и инициализации двумерного массива:

a) `int a[][] = new int[5][8];`
`int x=a.length-a[3].length;`

b) `int[][] a = new int[3][];`
`a[0]=null;`
`a[1]=new int[3];`
`a[2]=new int[4];`
`int x=a.length-a[1].length;`

c) `char a[][] = {{' ','&'},null,{' ',''},{'h','j'},{}};`
`int x=a.length*(a[0].length-a[2].length);`

2. Определите, какая характеристика двумерного массива (сумма, произведение, максимум, минимум, количество элементов, удовлетворяющих условию; по строкам, по столбцам или для всей матрицы) рассчитывается в данном фрагменте программы и заносится в `x`.

a) `int a[][] = new int[n][m];`
`int x[]=new int[m];`

```

    .   .   .
for(int i=0;i<m;i++){
    x[i]=0;
    for(int j=0;j<n;j++)
        x[i]+=a[j][i];}

```

b) `int a[][] = new int[n][m];`
`int x=a[0][0];`

```

    .   .   .
for(int i=0;i<a.length;i++){
    for(int j=0;j<a[i].length;j++)
        if(a[i][j]<x) x=a[i][j];}

```

c) `int a[][] = new int[n][m];`
`int x[]=new int[a.length];`

```

for(int i=0; i<a.length; i++){
    x[i]=0;
    for(int j=0; j<a[i].length; j++){
        if(a[i][j]==0) x[i]++;
    }
}

```

3. Напишите программу, вводящую последовательность положительных целых чисел произвольной длины, и печатающую минимальное значение индекса p , при котором $a_1 + a_2 + \dots + a_p > c$ (где a_i – элементы последовательности, c – целое число, которое задается в программе).
4. Напишите программу, вводящую последовательность вещественных чисел, и печатающую разность между максимальным и минимальным элементами последовательности.
5. Заполните массив случайными нечётными числами от 1 до 99, выведите его на экран в строку, а затем инвертируйте массив и выведите его элементы в столбец.
6. Для массива положительных целых чисел вычислите сумму произведений первого элемента на последний, второго на предпоследний и т. д.
7. Для массива действительных чисел вычислите значение многочлена степени n в точке x_0 , считая элементы массива коэффициентами этого многочлена.
8. Дан массив действительных чисел. Найти в массиве минимальный и максимальный элемент. Упорядочить элементы массива, расположенные между минимальным и максимальным по убыванию методом пузырька.
9. Дан одномерный числовой массив. Перестроить массив так, чтобы вначале подряд (в том же порядке) следовали отрицательные значения его элементов, затем нули, а затем положительные значения.
10. Заполните квадратную матрицу по заданному образцу:

$$\begin{pmatrix} 1 & 2 & 3 & K & 9 & 10 \\ 0 & 1 & 2 & K & 8 & 9 \\ 0 & 0 & 1 & K & 7 & 8 \\ M & M & M & O & M & M \\ 0 & 0 & 0 & K & 0 & \end{pmatrix}$$

11. Дана целая квадратная матрица n -го порядка. Определите, является ли она магическим квадратом, т.е. такой, в которой суммы элементов во всех строках и столбцах одинаковы.
12. Дана целая квадратная матрица n -го порядка. Упорядочить значения элементов главной диагонали по алгоритму вставки, а значения побочной диагонали – по алгоритму пузырька.
13. Осуществить обход матрицы по спирали по часовой стрелке, начиная с ее левого верхнего угла. Вывести элементы матрицы в порядке их обхода.
14. Преобразовать матрицу так, чтобы строки с нечетными индексами были упорядочены по убыванию, с четными – по возрастанию.
15. Дан двумерный целочисленный массив. Упорядочить строки матрицы по убыванию количества нечетных элементов в строке. Сортировка методом выбора.

5 ВВЕДЕНИЕ В КЛАССЫ, ОБЪЕКТЫ И МЕТОДЫ

Как известно, язык Java – это объектно-ориентированный язык программирования. В предыдущих главах, рассматривая примеры программ, мы никак особо не акцентировали внимание на объектной ориентированности языка, приближая стиль их написания к директивному стилю. Однако даже самая простая программа на Java должна состоять, по крайней мере, из одного класса и содержать методы. Поэтому в этой главе рассмотрим обзорно основные понятия объектно-ориентированного программирования (ООП), не углубляясь в концепцию объектно-ориентированного подхода к проектированию программ, которая будет подробно изложена позже.

Методология ООП основана на представлении программы в виде совокупности объектов, каждый из которых является реализацией определенного класса. В качестве объектов в программировании могут быть представлены конкретные предметы, а также абстрактные или реальные сущности. Это могут быть числа, строки, элементы пользовательского интерфейса (кнопка, переключатель, текстовое поле), шрифт, а также любые другие объекты, придуманные и реализованные программистом.

Описывая поведение какого-либо объекта, следует построить его модель. Выбрать наиболее значимые для написания программы характеристики и свойства объекта. Описание каждой модели производится в виде одного или нескольких классов. Руководствуясь определением класса, разработчик создает произвольное количество объектов, или экземпляров, данного класса. Класс и его объекты можно сравнить, соответственно, с чертежом и деталями – имея чертеж, можно произвести необходимое количество деталей. Таким образом, класс – это *шаблон* для объекта, а объект – это конкретный *экземпляр* класса.

Класс определяет общие для всех его объектов методы и свойства. Методы – это функции, реализующие некоторый алгоритм, который определяет взаимодействие объектов класса с внешней средой. Свойства представляют собой характеристики (атрибуты), присущие объектам (например, размер шрифта, название и др.).

5.1 Описание класса

В Java класс является основным программным элементом. Как вы уже успели заметить, все исполняемые программы Java являются классами. Однако сами классы не являются программами, так как в отличие от программ, представляющих собой последовательно выполняемые инструкции, элементы класса могут располагаться в произвольном порядке. До сих пор нами применялась только самая

элементарная форма классов, содержащая один единственный метод `main()`. Однако классы являются более мощными компонентами языка, чем представленные до сих пор.

Объявление класса состоит из заголовка и тела класса. Заголовок класса начинается со слова `class`, после которого записывается имя класса. Напомним, что в соответствии с соглашением об именовании, имена классов начинаются с прописной буквы (если имя содержит несколько слов, то каждое слово начинается с прописной буквы). Перед словом `class` могут быть записаны модификаторы класса (англ., `class modifiers`). Некоторые модификаторы будут рассмотрены позже.

В самом простом случае объявление класса имеет следующий вид:

```
class имя_класса
{
    /* Тело класса */
}
```

Тело класса может содержать объявления *членов класса*: переменных (полей), методов, вложенных классов, которые заключаются в фигурные скобки.

Полями называются переменные и константы, характеризующие состояние объекта или класса.

Методы – это функции (или подпрограммы) Java, которые объявляются только внутри классов и характеризуют поведение объектов. Таким образом, методы – это последовательности операторов, выполняющих какие-либо действия с полями для изменения состояния объекта.

Ниже записана общая форма определения класса:

```
class имя_класса {
    тип    переменная_экземпляра1;
    тип    переменная_экземпляра2;
    / /    . . .
    тип    переменная_экземпляраN;

    тип    имя_метода1 (список_параметров) {
    // тело метода
    }
    тип    имя_метода2 (список_параметров) {
    // тело метода
    }
    / /    . . .
    тип    имя_методаN (список_параметров) {
    // тело метода
    }
}
```

Определенные внутри класса поля называют *переменными экземпляра*, поскольку каждый экземпляр класса (объект) содержит собственные копии этих переменных. При описании переменной экземпляра указывается тип, затем, через пробел, имя и, может быть, начальное значение после знака равенства, которое можно записать константным выражением. Переменные могут быть ссылочного или примитивного типа. Описание переменной может начинаться с модификатора.

Ниже приведен пример класса, содержащего переменные примитивных типов:

```
class MyClass
{
    int i=5;//переменная экземпляра класса
    final int YEAR = 2014;//константа
    double d; //переменная экземпляра класса

    public void myMethod(int x, double y)//параметры метода
    {
        int a1,a2; //локальные переменные метода
        ...
    }
}
```

В примере создан класс с именем `MyClass`, в котором объявлены переменные и метод `myMethod()`. Метод содержит список параметров – переменные `x` и `y`. В теле метода объявлены переменные `a1` и `a2`. Обратите внимание на то, что переменные, объявленные внутри тела метода или входящие в список параметров, передаваемых в методы класса, являются локальными переменными. В нашем примере локальными переменными являются переменные `x`, `y`, `a1` и `a2`. Переменные же объявленные в теле класса, но вне методов – это глобальные переменные. Переменные с модификатором `final` являются константами. Модификатор `final` можно использовать для переменной, объявленной в методе, а также для параметра метода.

5.2 Области видимости переменных

Область видимости переменной определяется уровнем вложенности окружающих ее операторных скобок `{ }`. Самая внешняя пара скобок – скобки описания класса. Все переменные, описанные внутри этих и только этих скобок доступны всем элементам класса и называются *глобальными переменными*. Инициализировать глобальную переменную необязательно. Глобальным переменным гарантированно присваивается значение по умолчанию, если оно явно не указано. Значения по

умолчанию, присваиваемые глобальным переменным примитивных типов, приведены в табл.5.1.

Таблица 5.1 – Значения, присваиваемые глобальным переменным по умолчанию

<i>Примитивный тип</i>	<i>Значение по умолчанию</i>
boolean	false
char	'\u0000' (null)
byte	(byte) 0
short	(short) 0
int	0
long	0L
float	0.0f
double	0.0d

Значение по умолчанию гарантируются только глобальным переменным. В программах, рассматриваемых нами ранее, хотя и был объявлен класс, но объявление и инициализация переменных осуществлялась в теле метода `main()`. Переменная, описанная в блоке описания метода, называется *локальной*. Область ее видимости ограничена этим блоком, а время жизни – временем действия метода. Поэтому если такую локальную переменную использовать в методе заранее ее не проинициализировав, то компилятор выдаст соответствующую ошибку. Локальная переменная инициализируется обязательно.

Локальная переменная *скрывает* одноименную глобальную переменную.

5.3 Создание объектов

Рассмотрим упрощенный пример класса, описывающего поведение объекта – студент.

```
class Student
{
    //Объявление переменных
    int groupId ;      //номер группы
    String firstName; //имя студента
    String lastName;  //фамилия студента
    // Прочие поля...

    //Методы
    void setGroup(int newId)
```

```

{
    // Метод переводит студента в другую группу
    groupId = newId;
}
void getInfo()
{
    // Метод выводит информацию о студенте на экран
    System.out.println(firstName+" "+lastName+" "+groupId);
}
// Прочие методы. . .
}

```

Класс содержит описание трех переменных: номер группы `groupId` имя `firstName` и фамилия `lastName` студента и два метода `setGroup()` и `getInfo()` манипулирующие ими. Как видно в классе отсутствует метод `main()` и поэтому его невозможно исполнить. Но это и не нужно, так как классы рассчитаны на многократное использование другими классами в дальнейшем.

После того как описание класса закончено, можно создавать конкретные объекты, экземпляры описанного класса. Создание объектов производится в три этапа (подобно описанию массивов). Сначала объявляются ссылки на объекты, например:

```
Student stud1, stud2;
```

Записывается имя класса, и через пробел перечисляются переменные – ссылки на экземпляры класса. В этой записи наглядно видно, что созданный класс `Student` – это описание нового ссылочного типа данных. Все переменные, тип которых определен каким-либо классом, называются *ссылочными переменными*.

На втором этапе определяются сами объекты, под них с помощью операции `new` выделяется оперативная память и ссылка получает адрес этого участка в качестве своего значения.

```
stud1 = new Student();
stud2 = new Student();
```

После операции `new` записывается конструктор `Student()`. *Конструктор класса* (англ., class constructor) представляет собой снабженный параметрами именованный блок инструкций, который автоматически вызывается при создании объекта класса и выполняет действия по инициализации объекта. Оператор `new` обращается именно к этому блоку, в данном случае, с пустым списком параметров. Конструктор – обязательный элемент любого класса. Однако, как вы можете видеть, в классе `Student` конструктора нет. Дело в том, что к классу, не описавшему ни одного конструктора, компилятор

автоматически добавляет конструктор с пустым набором инструкций и пустым списком параметров. Это так называемый *конструктор по умолчанию*.

Переменная, связанная с объектом, является, по сути, именованным указателем на занимаемую этим объектом область памяти, которая доступна, пока существует хотя бы один такой указатель. Разорвать связь ссылочной переменной с экземпляром можно, переведя ее в состояние `null`, которое освобождает ее от связи с каким-либо объектом:

```
stud1 = null;
```

После создания экземпляров класса, можно обращаться к его элементам — переменным и методам — экземпляров. Как правило, до создания экземпляра переменные и методы класса недостижимы.

На этапе инициализации объектам можно присваивать начальные значения и вызывать методы. Поскольку имена переменных экземпляра и методов у всех объектов одинаковы, их нужно уточнять именем ссылки на объект. Для доступа к этим переменным применяется *оператор точка* (`.`). Этот оператор связывает имя объекта с именем переменной экземпляра, например:

```
stud1.groupId= 11;
stud1.lastName ="Петров";
stud2.lastName ="Сидорчук";
stud1.setGroup(12);
stud2.getInfo(12);
```

Обычно экземпляры классов создаются в методах других классов. Рассмотрим пример использования класса `Student`. Для этого опишем новый класс `StudentDemo`.

```
class StudentDemo{
public static void main(String[] args){
//Создадим экземпляр класса Student
Student stud1=new Student();
stud1.groupId=11;
stud1.firstName ="Сергей";
stud1.lastName ="Сидорчук";
//Вызовем метод getInfo(), выводящий информацию о
//студенте на экран
stud1. getInfo();
//Переведем студента в другую группу
stud1.setGroup(12);
// Выведем обновленную информацию
stud1. getInfo();
}}
```

В результате запуска метода `main()` класса `StudentDemo` на экран будет дважды выведена информация о студенте. Первый раз с номером группы – 11, второй раз уже с новым номером группы – 12.

В классе `StudentDemo`, можно создать сколь угодно много объектов класса `Student`. Можно даже создать массив таких объектов. Каждый созданный объект, будет содержать собственную копию каждой переменной экземпляра, определенной классом. Таким образом, каждый объект класса `Student` будет содержать собственные копии переменных экземпляра `groupId`, `firstName` и `lastName`. У всех объектов будет одинаковый набор атрибутов, но их значения могут быть различными и их можно менять. Изменение переменных экземпляра позволяет изменять и поведение методов экземпляра. В этом состоит одно из важных преимуществ объектного подхода.

Обратите внимание, что в примере, приведенном выше, объект был нами создан, но не удален в конце программы. Дело в том, что в Java нет прямого способа удаления объектов из памяти и, следовательно, нет специального оператора уничтожения объектов. Освобождение памяти от неиспользуемых объектов выполняется встроенным в виртуальную машину сборщиком мусора.

Сборщик мусора (англ. *garbage collector*) – это программа, которая разыскивает в памяти виртуальной машины объекты, свободные от ссылок на них, и удаляет их оттуда. Сборщик мусора периодически запускается виртуальной машиной без вмешательства программиста.

5.4 Методы класса

С самых первых программ Java, рассмотренных нами, мы сталкивались с методами. В качестве примера можно привести методы класса `Scanner` для ввода данных – `nextInt()`, `nextDouble()` или методы класса `Math` – `random()`, `pow()`, `sqrt()`, не говоря уже о методе `main()`, который должен присутствовать в любом исполняемом приложении. Пришло время рассмотреть подробнее, как объявляются и вызываются методы в Java и научиться писать свои собственные методы.

Ранее мы определили методы, как функции (или подпрограммы), которые объявляются только внутри классов и характеризуют поведение объектов. Для того чтобы получить более полное представление о методах, целесообразно разобраться в том, что в программировании понимается под функцией.

Функция – это часть программы, имеющая собственное имя. Основное назначение функции – уменьшить сложность программы и общее количество кода. Так, часто повторяющуюся законченную часть кода, решающую небольшую задачу, можно оформить в виде отдельной

функции (подпрограммы) и потом использовать в программе несколько раз, вызывая по имени²⁴.

Вызов функции – это отдельная команда, по которой начинают выполняться операторы, из которых функция состоит. Если функция при выполнении возвращает значение, то команду вызова функции вполне можно использовать в выражениях совместно с другими операциями. Интересно, что однажды написанная функция может быть включена в библиотеку функций и использоваться в программах других программистов.

Метод – это функция, являющаяся частью некоторого класса, которая может выполнять операции над данными этого класса. Так как в Java вся программа состоит только из классов, то и функции могут описываться только внутри них. Поэтому все функции в Java являются методами.

5.4.1 Объявление метода

Для того чтобы использовать в программе собственный метод, его необходимо объявить. Общая форма объявления метода имеет следующий вид:

```
тип имя_метода (список параметров)
{
    //тело метода
}
```

Описание метода может начинаться с модификаторов. При объявлении метода необходимо указать тип значения, которое будет возвращено после выполнения метода в программу. Тип может быть любым как примитивным, так и ссылочным. Если значение возвращать не нужно, то указывается ключевое слово `void`. Затем идет произвольный идентификатор – имя метода. После имени метода в круглых скобках указывается список параметров (он может быть и пустым). В списке параметров через запятую перечисляются тип и идентификатор каждого параметра. Далее в фигурных скобках записываются операторы, составляющие тело метода. Даже если метод не выполняет никаких действий и операторы отсутствуют, его тело все равно должно описываться парой пустых фигурных скобок. В теле метода не может быть описан другой метод, так как в Java нет вложенных процедур и функций.

Параметры – это данные, которые нужно передать в метод, чтобы корректно выполнялся его программный код. Например, метод, вычисляющий площадь круга, должен получить в качестве параметра радиус круга. А метод, который находит сумму элементов массива `int[]`

²⁴ Прием, при котором большую программу разбивают на подпрограммы, называют *декомпозицией*.

а, больших некоторого целого числа b, должен получить одномерный массив и целое число. Заголовок такого метода может иметь следующий вид:

```
int summa(int[] a, int b)
{
    //тело метода
}
```

Обратите внимание, что в отличие от объявления переменных, в списке параметров метода запрещается указывать несколько идентификаторов для одного типа:

```
void myMethod (double x, y) // неверная запись!
void myMethod (double x, double y) //верная запись
```

Кроме этого, в методе запрещены одноименные параметры и создание локальных переменных, имена, которых совпадают с именами параметров.

В классе можно описывать методы с одинаковыми именами, но разными параметрами. Такое дублирование методов называется *перегрузкой* (англ., *overloading*) и является возможным, потому что компилятор различает методы не по имени, а по сигнатуре.

Сигнатура (англ., *signature*) определяется именем метода и его параметрами: их количеством, типом и порядком следования. В классе запрещено создание двух методов с одинаковыми сигнатурами. Тип возвращаемого значения не входит в сигнатуру метода, значит, методы не могут различаться только типом результата их работы. Рассмотрим пример класса, содержащего методы с одинаковыми именами:

```
class Point
{
    void get() {...}
    void get(int x) {...}
    void get(int x, double y) {...}
    void get(double x, int y) {...}
}
```

Класс Point объявлен корректно. Все методы, хотя и имеют одинаковые имена, но различаются параметрами, а значит, имеют разную сигнатуру. Например, следующие пары методов несовместимы друг с другом в одном классе:

```
void get() {}
int get() {}
void get(int x) {}
double get(int y) {}
```

Методы отличаются типом возвращаемого значения, которое не входит в определение сигнатуры. Стало быть, данные пары методов с одинаковыми сигнатурами.

И последнее, если в заголовке метода указан тип возвращаемого значения, а не `void`, то в теле метода обязательно должен находиться оператор `return`, после которого через пробел указывается возвращаемое значение или выражение, совместимое по типу с типом возвращаемого значения. Оператор `return` прерывает работу метода и передает результат выполнения указанного выражения основной программе – в то место, откуда метод был вызван. В методе без возвращаемого значения (`void`) также можно использовать выражение `return` без каких либо аргументов. Его можно указать в любом месте метода, в этой точке выполнение метода будет завершено.

Используя рассмотренный выше синтаксис, попробуем в качестве примера создать метод для вычисления площади трапеции. При разработке собственного метода важно определить входные и выходные данные. Входные данные – это параметры метода. Выходные данные – это его возвращаемое значение.

Как известно, площадь трапеции равна произведению полусуммы ее оснований на высоту: $S = \frac{a+b}{2}h$. Входными данными для рассматриваемого метода будут, очевидно, длины двух оснований и высота трапеции. Выходные данные (результат) – вещественное число, представляющее собой площадь трапеции.

Следует отметить, что в объектно-ориентированной концепции методы могут получать данные для работы, обращаясь к атрибутам своего класса, а результат их работы может заключаться в изменении этих атрибутов. Мы несколько отойдем от объектно-ориентированной парадигмы и на данном этапе изучения языка Java, будем рассматривать методы как классические функции – подпрограммы.

Простейший вариант метода будет выглядеть следующим образом:

```
double areaTrapezoid (double a, double b, double h)
{return h*((a+b)/2);}
```

Создадим класс `MyClass` и добавим в него описание метода `areaTrapezoid()`. Данный класс является исполняемой программой, поэтому в нем будет находиться еще и метод `main()`, из которого при запуске программы нужно будет вызвать метод `areaTrapezoid()`. Для того, чтобы такой вызов был возможным, добавим в заголовке метода `areaTrapezoid()` модификатор `static` (он будет изучен нами позднее). Описание метода `areaTrapezoid()` должно находиться внутри класса, но не внутри метода `main()`, а на одном уровне с ним:

```

class MyClass {
    static double    areaTrapezoid (double a, double b,
double h)
        {return h*((a+b)/2);}

    public static void main(String[] args)
        { // вызов метода areaTrapezoid()
        }}

```

В результате класс `MyClass` содержит два метода: `main()`, выполняющийся при запуске программы и `areaTrapezoid()`, который необходимо вызвать.

5.4.2 Вызов метода

Чтобы вызвать метод из другого метода того же класса, необходимо указать его имя, а затем в скобках список *аргументов* (фактических параметров). Если метод не требует аргументов, то скобки все равно ставятся.

Например, метод `areaTrapezoid()` можно вызвать, передав ему в качестве аргументов три вещественных числа 5.5, 10.2 и 4.1, следующим образом:

```
double s = areaTrapezoid (5.5, 10.5, 4.0);
```

Как видите, вызов метода выполняется в правой части оператора присваивания.левой частью этого оператора является переменная `s`, которая будет принимать значение, возвращенное методом `areaTrapezoid()`. В данном случае возвращаемое значение будет равно 32.0 и именно это значение сохранится в переменной `s`.

При вызове в метод передается список аргументов, а в заголовке метода указывается список параметров. Следует различать эти понятия.

Параметр – это определенная в методе переменная, которая принимает значение при вызове метода. Например, в методе `areaTrapezoid()` параметром являются переменные `a`, `b` и `h`.

*Аргумент*²⁵ – это значение, передаваемое методу при его вызове. Например, в примере методу в качестве аргументов передаются значения 5.5, 10.2 и 4.1. Внутри метода `areaTrapezoid()` параметры получают эти значения в том порядке, в котором они записаны: `a=5.5`, `b=10.2`, `h=4.1`. *Между параметрами и аргументами должно быть соответствие по порядку следования, по количеству и по типу.*

²⁵ За параметрами и аргументами в программировании также закрепились названия *формальные* и *фактические* параметры соответственно.

Обратите внимание на то, что и переменная, принимающая возвращенное методом значение (в примере `s`), должна быть совместима с возвращаемым типом, указанным в методе.

Вызов метода можно использовать как операцию, комбинируя с другими операциями в выражении, например:

```
double s = areaTrapezoid (5.5, 10.5, 4.0)+10;
```

Вызов метода можно было осуществить и без использования переменной `s`, например, внутри метода `printf()`:

```
System.out.printf("Площадь=%5.1f", areaTrapezoid(5.5,  
10.5, 4.0));
```

В этом случае метод `areaTrapezoid()` будет вызван автоматически при вызове метода `printf()`, а возвращаемое им значение будет передано методу `printf()`.

Чтобы вызвать метод другого класса, необходимо иметь объект этого класса. При вызове имя метода указывается через точку после имени объекта. Общая форма вызова метода имеет следующий вид:

```
имя_объекта.имя_метода(список_аргументов);
```

Именно, так мы и вызывали метод `getInfo()` для объекта экземпляра класса `Student`:

```
stud1. getInfo()
```

5.5 Статические элементы

Итак, методы вызываются для объектов. Однако чтобы создать объект, необходимо запустить программу и метод `public static void main(String[] args)`, а это означает, что есть метод, который доступен еще до создания каких бы то ни было объектов. Обратите внимание, что в заголовке метода указаны модификаторы `public` и `static`. Дело в том, что виртуальная машина Java для исполнения класса-программы считывает код класса в память и разыскивает в нем метод с таким названием и с такими модификаторами. Кроме того, на этом этапе считываются только методы и переменные, отмеченные модификатором `static` – *статические элементы* класса.

Статические элементы принадлежат классу, а не экземпляру класса.

Для статических переменных, называемых также *переменными класса*, выделяется только одна ячейка памяти, общая для всех экземпляров. Изменение статической переменной в одном экземпляре класса мгновенно приведет к ее изменению во всех экземплярах. К статическим переменным можно обращаться по имени класса, а не только по имени экземпляра, причем это можно делать, даже если не создан ни один экземпляр класса.

Для статических методов исполняющая система Java всегда создает в памяти только одну копию машинного кода метода. Статические методы выполняются сразу во всех экземплярах класса. Они могут выполняться, даже если не создан ни один экземпляр класса. Статические методы имеют доступ только к статическим переменным класса, обращение из статического метода к нестатической переменной класса является ошибкой. Чтобы вызвать статический метод достаточно уточнить имя метода именем класса (а не именем объекта). Некоторые стандартные статические методы мы уже вызывали так в программах рассмотренных ранее. Например, методы класса `Math` – `Math.abs(x)`, `Math.sqrt(x)` или метод `System.out.println()`.

Если статический метод вызван в классе, в котором он описан, то его можно вызывать просто по имени. Именно так, по имени, был вызван метод `areaTrapezoid()` в рассмотренном ранее примере.

Модификатором `static` можно отметить и блок операторов, размещенный на уровне описания класса. В этом случае он как статический элемент класса будет выполнен виртуальной машиной сразу же после загрузки класса.

Назначение других модификаторов, используемых в объявлениях членов класса, будет рассмотрено позже.

Отметим только, что добавлением модификатора доступа `public` к описанию члена класса, позволяет сделать его открытым, то есть доступ к таким членам класса (полям или методам) разрешен любому классу.

В противоположность открытости можно объявить некоторые члены класса закрытыми, записав вместо слова `public` модификатор доступа `private`. К таким методам и переменным разрешается доступ только внутри класса, в котором они объявлены.

Если модификатор доступа не указывается при объявлении переменных или методов, то доступ к ним разрешается для любых классов того же пакета, в котором размещается класс, где объявляются эти методы и переменные.

В ближайших главах по мере изучения основ языка программирования Java будем в программах-примерах описывать статические методы, и вызывать их в методе `main()` просто по имени. Это будет временно, и до тех пор, пока не будет подробно изложена концепция объектно-ориентированного подхода.

5.6 Способы передачи аргументов

В программировании есть два понятия: передача в подпрограмму аргумента *по ссылке* и *по значению*.

При *передаче по значению* – аргумент копируется в формальный параметр подпрограммы. Следовательно, если параметр изменяет свое значение в подпрограмме, то изменение аргумента происходить не будет.

При *передаче по ссылке* – параметру передается ссылка на аргумент (а не его значение). Внутри подпрограммы эта ссылка используется для обращения к реальному аргументу, указанному в вызове. Это означает, что изменения, выполненные в параметре, будут влиять на аргумент, использованный в вызове подпрограммы.

Все аргументы в Java передаются по значению. Однако конкретный эффект будет зависеть от того, передается ли аргумент примитивного или ссылочного типа.

В случае, когда аргумент – примитивный тип, передача по значению означает то, что метод не может изменить оригинальное значение аргумента в программе. Рассмотрим пример.

```
class MyClass{
    static void add(int i){
        i+=4;
    }
    public static void main (String[] args)
    {
        int i=10;
        System.out.println("i перед вызовом:"+i);
        add(i);
        System.out.println("i после вызова:"+i);
    } }
```

Результат

```
i перед вызовом:10
i после вызова:10
```

Как видите, выполняемые внутри метода `add()` операции не влияют на значение переменной `i`, оно не изменилось после вызова метода.

Когда же аргумент – ссылка на объект, при передаче этой ссылки методу, принимающий ее параметр будет ссылаться на тот же объект, на который ссылается аргумент. По сути, это означает, что объекты передаются так, как если бы они передавались методам по ссылке. Изменения объекта внутри метода влияют на объект, использованный в качестве аргумента. Рассмотрим несколько примеров передачи аргументов ссылочного типа.

```
class MyClass{
    static void add(int[] a){
        for(int i=0;i<a.length;i++)
            if (a[i]>0) a[i]=0;
    }
    public static void main (String[] args){
```

```

int [] m={1,2,-3,0,-4};
System.out.println("Массив перед вызовом метода:");
for(int x : m) System.out.printf("%3d",x);
System.out.println();
add(m);
System.out.println("Массив после вызова метода:");
for(int x : m) System.out.printf("%3d",x);
} }

```

Результат

Массив перед вызовом метода:

```
1  2 -3  0 -4
```

Массив после вызова метода:

```
0  0 -3  0 -4
```

В примере в метод передается аргумент ссылочного типа — одномерный массив. Как видите, в данном случае действия внутри метода `add()` влияют на массив `m[]`. Элементы массива поменяли свое значение.

Теперь рассмотрим другой пример передачи объекта.

```

class My {
    int i;
    static void changeIt(My value) {
        value.i= 10;
    }
    public static void main(String[] arg)
    { My obj = new My();
      obj.i = 20;
      System.out.printf("Значение  obj.i до  вызова  метода%3d\n",
obj.i);
      changeIt(obj);
      System.out.printf("Значение obj.i после вызова метода%3d",
obj.i);
    }}

```

Результат

Значение `obj.i` до вызова метода 20

Значение `obj.i` после вызова метода 10

Если программисту требуется, чтобы вызванный метод изменил переданные ему значения переменных примитивных типов, то эти переменные можно также обернуть специальным объектом (или использовать массив) и передать эту "обертку" в качестве аргумента.

Обратите внимание, несмотря на то, что тип `String` — ссылочный тип, изменение значения строковой переменной в теле метода не влияет на значение этой же переменной снаружи метода:

```

public class My {
    static void changeIt(String value) {
        value = new String("Hello!");
    }
}

```

```

public static void main(String[] args){
String test = new String("HelloWorld!");
changeIt(test);
System.out.println("Новое значение: "+ test);
}}

```

Результат

Новое значение: HelloWorld!

Задача 5.1. Создайте класс, содержащий метод, который с точностью до $\text{Eps}=1\cdot 10^{-4}$ рассчитывает таблицу значений функции вещественной переменной x .

$$\ln x = (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \dots + (-1)^{n+1} \frac{(x-1)^n}{n}, \quad 0 < x \leq 2,$$

Результаты работы сравните с результатом расчета соответствующего метода класса Math.

```

public class MyClass {
static final double eps=1e-4;

public static double mylog(double x) {
double sum=x-1,c=x-1;
int n=1;
while(Math.abs(c/n)>eps){
c*=-(x-1); sum+=c/++n; }
return sum;
}

public static void main(String[] args) {
System.out.printf(" ----- %n");
System.out.printf("| x | mylog(x) | log(x) | %n");
System.out.printf(" ----- %n");
for(double x=0.1;x<=2;x+=(2-0.1)/5){
System.out.printf("| %8.4f | %8.4f | %8.4f | %n", x, mylog(x),
Math.log(x)); }
System.out.printf(" ----- %n");
}}

```

Результат:

x	my_log(x)	log(x)	

0,1000	-2,3019	-2,3026	
0,4800	-0,7339	-0,7340	
0,8600	-0,1508	-0,1508	
1,2400	0,2151	0,2151	
1,6200	0,4824	0,4824	
2,0000	0,6931	0,6931	

5.7 Список аргументов переменной длины

При вызове метода можно передавать переменное количество аргументов – `vararg` (сокращение от `variable-length arguments` – список аргументов переменной длины). Метод, который принимает переменное количество аргументов, называют *методом с переменным количеством аргументов*. Такие методы удобны тогда, когда количество аргументов заранее не известно.

Для указания списка аргументов переменной длины используют три точки (`...`). Например, вот так бы выглядел заголовок метода с переменным количеством аргументов для расчета среднего арифметического целых аргументов:

```
static double average (int ... v) {}
```

Эта синтаксическая конструкция указывает компилятору, что метод `average()` может вызываться без аргументов или с несколькими аргументами. Параметр `v` неявно объявляется как массив типа `int[]`. Таким образом, внутри метода `average()` доступ к параметру `v` осуществляется с использованием синтаксиса обычного массива.

Задача 5.2. *Опишите метод с переменным количеством аргументом, вычисляющий среднее арифметическое значение целых аргументов.*

Объяснение: в методе `main()` метод `average()` вызывается с различным количеством аргументов, в том числе и вовсе без аргументов. Аргументы автоматически помещаются в массив и передаются параметру `v`. В случае отсутствия аргументов длина массива равна нулю. Поэтому в этом случае при нахождении среднего арифметического значения возвращается значение `NaN` – не число.

```
class MyClass {
    static double average (int...v)
    { double s=0;
      for(int x  :  v) s+=x;
      return s/v.length;
    }
    public static void main(String args[])
    {
        System.out.printf("Без аргументов:s=%5.2f\n",average());
        System.out.printf("C аргументом
10:s=%5.2f\n",average(10));
        System.out.printf("C аргументами 10,20,30:s=%5.2f\n",
        average(10,20,30));
    }
}
```

Результат

Без аргументов:s= NaN
С аргументом 10:s=10,00
С аргументами 10,20,30:s=20,00

Список параметров в заголовке метода может содержать как параметр с переменным количеством аргументов, так и обычные параметры. Однако параметр с переменным количеством аргументов должен быть последним параметром в списке и единственным в методе.

Например, следующие объявления методов являются ошибочными:

```
//Неверная запись!  
void myMethod(int a, int b, int...v, boolean f) {}  
void myMethod(int a, int b, int...v, double...w ) {}
```

В первом случае предпринимается попытка объявления обычного параметра после параметра с переменным количеством аргументов, что недопустимо. А во втором – попытка объявления второго параметра с переменным количеством аргументов, что тоже запрещено.

Следующее объявление метода считается вполне корректным:

```
void myMethod(int a, int b, double c, int...v) {}
```

При вызове такого метода, первые три аргумента будут соответствовать первым трем параметрам *a, b, c*. Все остальные аргументы считаются принадлежащими параметру *v*.

Метод с переменным количеством аргументов можно перегружать. В этом случае перегруженные варианты должны различаться либо типом, либо количеством параметров. Например, в следующем классе все перегруженные варианты метода *get* записаны корректно:

```
class MyClass {  
    void get(int...v){}  
    void get(int n){}  
    void get(double...v){}  
    void get(int n, double...v){}  
}
```

Обратите внимание, что метод с переменным количеством аргументов может быть перегружен также методом, который не поддерживает эту возможность. Например, в приведенном классе *MyClass* метод *get(int...v)* перегружен методом *get(int n)*. Если при вызове метода указать один целочисленный аргумент, то будет вызван метод *get(int n)*. В случае передачи методу двух и более аргументов типа *int* программа будет использовать версию метода *get(int...v)* с переменным количеством аргументов.

Будьте внимательны, используя перегруженные варианты методов с переменным количеством аргументов, так как при их вызове могут

возникать неопределенности. Так, например, если будет вызван метод `get()` без аргументов, то это приведет к ошибке, так как данный вызов может быть преобразован в обращение к методу `get(int...v)` или `get(double...v)`. Оба варианта допустимы. Поэтому вызов принципиально неоднозначен.

5.8 Конструкторы

Вспомним, как создаются объекты в Java. Основной способ создания экземпляров классов – это применение оператора `new` с конструктором класса, например:

```
Student stud = new Student();
```

Обратите внимание на то, что запись конструктора похожа на вызов метода. Однако конструктор – не метод. Конструкторы имеют только одно назначение – создать экземпляр класса. Тогда, как назначение методов гораздо более общее – выполнить программный код. Конструктор не является методом и даже не считается членом класса.

Ниже перечислены некоторые особенности конструктора:

- конструктор имеется в любом классе. Даже если он не был описан в классе, компилятор сам создаст *конструктор по умолчанию* без параметров;

- имя конструктора всегда совпадает с именем класса, в котором он описан;

- конструктор не может быть вызван иначе как оператором `new` и выполняется автоматически при создании экземпляра класса, но до начала использования создаваемого объекта;

- конструктор не возвращает никакого значения. В его описании не пишется даже служебное слово `void`.

В классе может быть описано несколько конструкторов. Поскольку у них одно и то же имя, совпадающее с именем класса, то они должны отличаться типом и/или количеством параметров.

Обратите внимание на особенность создания конструктора по умолчанию. Если конструктор не был создан в классе, то он создается автоматически. Однако если в классе программистом уже описан один или несколько конструкторов (с аргументами или без), то компилятор больше не будет создавать конструктор по умолчанию. Рассмотрим пример.

Создадим простой класс с несколькими конструкторами:

```
class My{
    int i;
    My(int a) {}//конструктор 1
    My (double b) {} //конструктор 2

    public static void main(String[] args){
```

```
        My h = new My(); //Ошибка!  
    }  
}
```

Программный код ошибочный: после того как конструкторы описаны, компилятор не создаст конструктор по умолчанию, а значит он не сможет найти конструктор, подходящий по описанию.

Следующая запись верна – компилятор запустит первый конструктор с целочисленным параметром.

```
        My h = new My(5); //Верно
```

В классе можно создавать конструкторы с аргументами переменной длины. Обратите внимание, что если в классе описан конструктор, имеющий аргумент переменной длины, то вызов конструктора по умолчанию ошибкой не будет, так как аргумент переменной длины может быть пустым. Например:

```
class My{  
    int i;  
    My(int...a) {}//конструктор 1  
    My (double b) {} //конструктор 2  
    public static void main(String[] args){  
        My h = new My();  
    }  
}
```

Компиляция и запуск такой программы пройдут успешно.

Иногда для того, чтобы сослаться на текущий объект в методах и конструкторах используют ключевое слово `this`.

В методах `this` использует для того, чтобы получить ссылку на экземпляр класса выполняющего этот метод. Статические методы не используют ключевое слово `this`, так как они не принадлежат экземпляру класса, поэтому `this` некуда ссылаться.

Рассмотрим несколько примеров применения `this` в конструкторах. В следующем классе `MyClass` имеются два конструктора с различными входными параметрами. В первом конструкторе выполняется вызов второго конструктора данного класса с использованием ключевого слова `this`:

```
class MyClass{  
    int a;  
    int b;  
  
    MyClass() {  
        this(10,5);  
    }  
  
    MyClass(int a, int b) {  
        this.a=a;  
        this.b=b;  
    }  
}
```

```

public static void main(String[] args){
    MyClass h = new MyClass();
    System.out.printf("a+b=%3d",h.a+h.b);
}
}

```

Результатом выполнения программы будет строка: a+b= 15.

В методе `main()` создается объект `h` с помощью конструктора без параметров, который в свою очередь вызывает конструктор с двумя целыми параметрами `a` и `b`, передавая для них аргументы 10 и 5 соответственно.

Во втором конструкторе ключевое слово `this` используется для получения доступа к переменным экземпляра `a` и `b` с такими же именами, что и у локальных переменных `a` и `b`. Без использования ключевого слова `this` формальные параметры в данном случае скрыли бы переменные экземпляра.

Рассмотрим еще один пример программы, в которой создается класс `Rect`, описывающий ширину и высоту прямоугольника и включающий в себя метод экземпляра `area()`, возвращающий площадь прямоугольника и метод класса `perimeter()`, вычисляющий периметр.

```

class Rect {
    int width;
    int height;

    //параметризованный конструктор
    Rect(int width, int height) {
        this.width = width;
        this.height = height;
    }
    // метод экземпляра
    public int area() {
        return width * height;
    }

    // метод класса
    static int perimeter(Rect h) {
        return 2*(h.width + h.height);
    }
}

class UseRect {
    public static void main(String[] args) {
        Rect r = new Rect(4, 5);
        System.out.printf("Площадь:%d\n", r.area());
        System.out.printf("Периметр:%d", Rect.perimeter(r));
    }
}

```


В этом варианте написания конструктора `Rect()` имена параметров совпадают с именами переменных экземпляра, а, следовательно, скрывают их. Но для “обнаружения” скрытых переменных служит ключевое слово `this`. Вторым интересным моментом в этом примере является иллюстрация использования *метода класса*. Второй из двух методов класса – `perimetr()` описан с ключевым словом `static`, что делает его методом класса. Такой метод не может быть вызван от конкретного объекта с помощью оператора точка (`.`), ему не передается указатель `this` на экземпляр, а само это ключевое слово не может быть использовано в его теле. Поэтому методу в качестве параметра следует явно передать ссылку на объект. Вызывается статический метод просто по имени. Так как в примере метод вызывается из другого класса, то его имя уточняется именем класса, в котором он описан: `Rect.perimetr(r)`.

5.9 Класс Stack

В качестве примера, иллюстрирующего работу с классами, рассмотрим создание практического приложения, реализующего стек. *Стек* представляет собой структуру данных, работа с которой осуществляется по принципу LIFO (англ., last-in first-out – последним пришел – первым вышел). Добавить элемент в стек можно в любой момент, удаляется же только элемент, добавленный последним.

Стек обязательно поддерживает два основных метода:

`push()`: помещает элемент на вершину стека.

`pop()`: извлекает элемент из стека, если стек пуст, выдается сообщение об ошибке.

Рассмотрим пример реализации стека целых чисел с помощью массива.

```
class Stack{
    int s[];
    int t;

    // конструктор
    public Stack(int size){
        s = new int [size];
        t = -1;
    }

    //методы экземпляра класса
    boolean isEmpty(){
        return (t == -1);
    }

    void push(int elem){
```

```

        if(t+1 < s.length)
            s[++t] = elem;
    }
    int pop(){
        if(isEmpty()){
            System.out.println("Стек пуст.");
            return 0;}
        return s[t--];
    }
}

```

В классе `Stack` дополнительно реализован метод `isEmpty()`, возвращающий логическое значение `true` если стек пуст. Класс содержит два атрибута: массив целых чисел – элементы стека; и целую переменную `t`, индексирующую массив (она всегда содержит индекс вершины стека). Начальное значение переменной `t` равно `-1`. Это признак того, что стек пуст.

Задача 5.3. Дан массив, состоящий из n целых элементов. Необходимо сначала ввести элементы массива, а затем перевернуть массив и вывести результат на экран.

Объяснение: данную задачу можно просто реализовать с помощью стека, так как метод `pop()` извлекает элемент, помещенный в стек последним.

```

import java.util.Scanner;
class StackDemo{
    static Stack stack;
    public static void main (String[] args)
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Введите количество элементов");
        int n = scan.nextInt();
        int[] a = new int [n];

        //создаем экземпляр класса Stack
        Stack stack = new Stack(a.length);

        //добавляем элементы в стек
        for (int i=0; i < a.length; i++) {
            System.out.print("элемент"+ i + ":");
            stack.push(scan.nextInt());
        }

        //извлекаем элементы из стека
    }
}

```

```
System.out.println("Переворот элементов:");  
for (int i=0; i < a.length; i++)  
System.out.println(stack.pop());  
}}
```

Результат:

```
Введите количество элементов  
5  
элемент0:8  
элемент1:9  
элемент2:7  
элемент3:1  
элемент4:2  
Переворот элементов:  
2  
1  
7  
9  
8
```

Упражнения к главе 5

1. Что неправильно в следующей программе? Исправьте ее.

```
public class SomethingIsWrong {  
    public static void main(String[] args) {  
        Rectangle myRect;  
        myRect.width = 40;  
        myRect.height = 50;  
        System.out.println("myRect's area is " + myRect.area());  
    }  
}
```

2. Фрагмент кода создает один объект `Point` и один объект `Rectangle`. Сколько ссылок к этим объектам существует после того, как код будет выполнен? Будет ли какой-нибудь из объектов кандидатом для «сборки мусора»?

...

```
Point point = new Point(2,4);  
Rectangle rectangle = new Rectangle(point, 20, 20);  
point = null;
```

...

3. Используя понятие сигнатуры метода, определите, какие методы несовместимы друг с другом в одном классе

```
class Complex {  
    void set(double re) {}  
    void set(double re, double im) {}  
    void set(Complex re) {}  
    void set(double x, double y) {}  
    public void set(Complex re) {}  
}
```

```
}
```

4. Определите, что будет выведено на экран при компиляции и запуске следующего кода:

```
public class Example{
    static int i;
    public static void main(String[] args){
        System.out.print(i);
    }
}
```

5. Определите, что будет выведено на экран при компиляции и запуске следующего кода:

```
public class MyClass {
    {System.out.print("1");}
    static{System.out.print("2");}
    MyClass(){System.out.print("3");}
    public static void main(String[] args) {
        System.out.print("4");
    }
}
```

6. Определите, что будет выведено на экран при компиляции и запуске следующего кода:

```
class My{
    int x,y;
    My(int x, int y){this.x=x; this.y=y;}
    My(int x){this(x,-2);}
    My(){this(1,2);}
    public static void main (String[] args){
        My a=new My(5);
        System.out.println((a.x+a.y)); }
}
```

7. Дан метод и три программных блока, каждый из которых обращается к указанному методу. Следует проверить верность написания каждого из этих блоков; если он верен – указать, что будет напечатано в качестве вывода.

```
static boolean flag (int a, int b){
    return (4*b!=a+b)?true:false;
}

. . .
1. int x=(flag (6,2))? 5:-5; System.out.println(x);
2. int x=2; double y=8.; System.out.println(flag(x,y));
3.int m=1; for(int i=3;i<6;i++)
    System.out.println(flag (i,m));
```

8. Имея класс с названием Point, напишите некоторый код, который создает экземпляр этого класса, инициализирует его две переменных, и затем выводит на экран значение каждой из переменных.

```
public class Point {
    public int x = 0;
    public int y = 0;
    public Point(int x, int y) {
        this.x = x;
```

```
this.y = y;  
}
```

9. Определите класс `Point`, задающий точку на плоскости. Класс должен содержать несколько конструкторов. Реализовать следующие методы для вычисления расстояния от точки до начала координат, расстояния между двумя точками, площади треугольника построенного по трем точкам.
10. Определите класс `Vector`, позволяющий выполнять над векторами на плоскости следующие операции: сложение, вычитание, умножение на число и вычисление скалярного произведения.
11. Создайте класс `Complex`, содержащий методы, позволяющие производить над комплексными числами арифметические операции сложения, вычитания, умножения и выводить их на экран.
12. Определите класс `Matrix` размерности $(m \times n)$. Класс должен содержать несколько конструкторов. Объявите массив объектов. Передайте объекты в метод, меняющий местами строки с максимальным и минимальным элементами k -го столбца. Создайте метод, который изменяет i -ю матрицу путем возведения ее в квадрат.

Чтобы понять рекурсию, надо понять рекурсию...

Автор неизвестен

6 РЕКУРСИЯ

Рекурсия – это такой способ организации обработки данных, при котором программа вызывает сама себя. Рекурсия называется *прямой* тогда, когда программа вызывает сама себя непосредственно или *косвенной* – когда программа вызывает саму себя опосредованно, через вызов других программ. Соответственно метод, который внутри своего тела вызывает сам себя, называется *рекурсивным методом*. Количество вложенных вызовов метода называется *глубиной рекурсии*.

Существует ряд задач, при решении которых обоснованным является использование именно рекурсивных методов. Например, вычисление различных рекуррентных последовательностей, в которых каждый следующий член, начиная с некоторого, определяется через предыдущие значения.

6.1 Стек вызовов

Чтобы понять, как работает рекурсивный метод, нужно знать, как распределяется память во время исполнения программы. В текущий момент времени может исполняться только один единственный метод из всей программы. Это значит, что, если метод `a()` вызывает в своем теле метод `b()`, а сам `a()` вызывается в `main()`, то при запуске программы управление сначала будет передано методу `main()`, затем методу `a()`, затем методу `b()`. Метод `b()` вернёт результат и управление в `a()`, `a()` вернет результат и управление в `main()`, и только потом будут выполняться команды, указанные в методе `main()` в строках, записанных после строки с вызовом `a()`.

Вся иерархия вызовов хранится в специальной области памяти, называемой *стеком вызовов*. Элементы в этот участок памяти добавляются по принципу LIFO: последний добавленный элемент должен быть извлечён первым. Когда метод вызывает сам себя, новым локальным переменным и параметрам выделяется место в стеке и код метода выполняется с этими новыми начальными значениями. При каждом возврате из рекурсивного вызова старые локальные переменные и параметры удаляются из стека, и выполнение продолжается с момента вызова внутри метода.

6.2 Рекурсия без ветвления

Напишем рекурсивную программу, вычисляющую факториал введенного натурального числа. Напомним, что факториал $n!$ целого

неотрицательного числа n задается следующим рекуррентным соотношением:

$$0! = 1,$$
$$n! = n \cdot (n - 1)! \text{ для } n > 0.$$

В задаче будем использовать рекурсивный метод, параметром которого является введенное натуральное число, а результатом – значение его факториала.

```
import java.util.Scanner;
class FactR {
    static long fact(int n) {
        return (n == 0)?1:n*fact(n-1);
    }
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Введите n:");
        System.out.printf("n!=%d", fact(scan.nextInt()));
    }
}
```

В приведенной программе метод `fact()` получает на вход целое число n и рекурсивно вычисляет его факториал. Рассмотрим процесс выполнения данной программы для $n=3$ более подробно (рис.6.1).

Так как число 3 отлично от нуля, метод попытается умножить число 3 на `fact(2)`, но последняя величина для ее вычисления требует вызова метода `fact()` с аргументом 2. В результате чего выполнение метода `fact()` с аргументом 3 приостановится, и программа приступит к выполнению этого же метода с меньшим на единицу значением аргумента. Вычисление `fact(2)` потребует нахождения `fact(1)`, что вызовет появление еще одного – третьего экземпляра метода `fact()`. Он, в свою очередь, вызовет `fact(0)`. К этому моменту стек вызовов метода накопит уже четыре вызова экземпляров метода с разными аргументами. При этом вычисления во всех вызванных ранее экземплярах метода приостановлены до завершения работы с последним.

Метод `fact()`, вызванный с нулевым аргументом, самостоятельно вычислит и вернет с помощью оператора `return` результат – число 1. Верхний элемент из стека вызовов методов после этого удалится, и возобновятся вычисления величины `fact(1)`. Этот процесс продолжится до тех пор, пока стек вызовов не станет пустым, что произойдет по завершению вычисления значения `fact(3)`. В итоге в метод `main()` будет возвращено значение 6. Описанный процесс хорошо иллюстрирует диаграмма, представленная на рис. 5.1. На ней наглядно видно, что при выполнении рекурсивного метода сначала происходит так называемое «рекурсивное погружение», а затем «возврат вычисленных результатов».

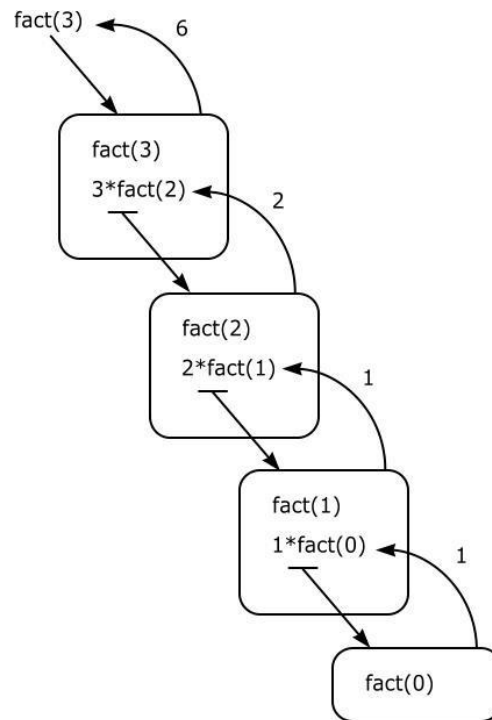


Рис.6.1 – Диаграмма стека вызовов рекурсивного метода `fact(3)`

Каждый рекурсивный метод должен иметь *условие завершения рекурсии*, в противном случае, он будет вызывать себя до тех пор, пока не переполнится стек вызовов. По условию завершения осуществляется возврат из рекурсивного метода без выполнения рекурсивного вызова. В методе `fact()` для этого была использована тернарная условная операция, но можно было использовать и оператор `if`.

Отметим, что алгоритм, реализованный в рекурсивной форме, всегда можно представить в итерационном виде, и наоборот. Так, в главе 3 нами уже была решена задача вычисления факториала натурального числа с использованием цикла.

В большинстве случаев рекурсивные версии программ выполняются несколько медленнее, чем их итерационные варианты. Это связано с дополнительными вызовами методов. Кроме того, поскольку параметры и локальные переменные сохраняются в стеке, а каждый новый вызов создает новые копии этих значений, то на каждый рекурсивный вызов требуется некоторое количество оперативной памяти компьютера, и при чрезмерно большой глубине рекурсии может наступить переполнение стека вызовов. Но, несмотря на это, существуют алгоритмы, реализация которых с помощью рекурсивных методов все же намного проще и понятнее, чем с помощью итераций.

Одним из отличий рекурсии от итерации является то, что в тело цикла с текущими значениями локальных переменных управление

передается один раз, а в тело рекурсивного метода два раза (может быть и более двух раз).

Задача 6.1 Напишите метод, который читает с клавиатуры положительные целые числа до ввода первого отрицательного числа, и печатает их в обратном порядке. Количество чисел заранее неизвестно.

Объяснение: напомним рекурсивный метод. Условием завершения рекурсивного метода будет условие ввода отрицательного числа. Диаграмма вызовов рекурсивного метода `revers()` при вводе последовательности из трех положительных чисел приведена на рис. 6.2. При рекурсивном вызове переменные вызывающего метода остаются в памяти и их значения становятся доступными после завершения рекурсивного вызова. Поэтому если в теле метода после строки с рекурсивным вызовом записать строку с оператором вывода на экран, то введенные значения будут выведены в обратном порядке.

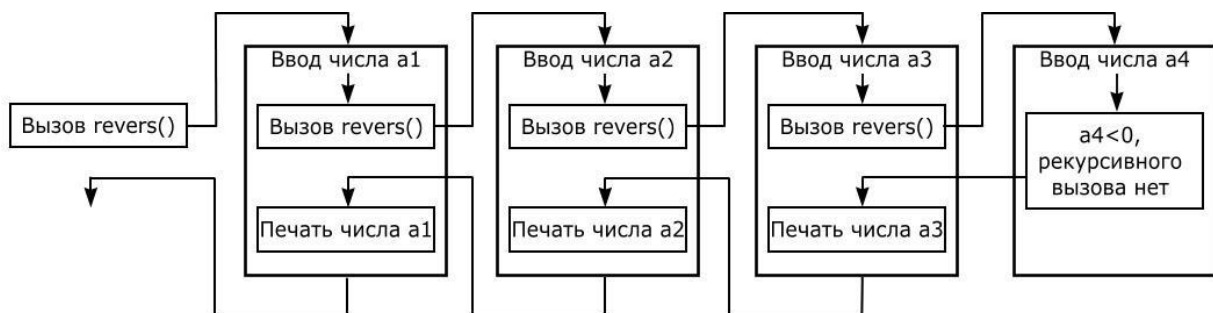


Рис.6.2 – Диаграмма вызовов рекурсивного метода `revers()`

```
import java.util.Scanner;
class Ex_6_1
{
    static void revers(){
        int a = new Scanner(System.in).nextInt();
        if (a<0) return;
        revers();
        System.out.printf("%d ",a);
    }
    public static void main (String[] args)
    {revers();}
}
```

Результат:

4
6

8
9
12
-5
12 9 8 6 4

Среди рекурсивных методов можно выделить методы, построенные по схеме *хвостовой рекурсии*. Такие методы вызывают себя только перед самым выходом из метода, т.е. оператор `return` в теле такого метода является самой последней инструкцией. К хвостовой рекурсии можно отнести рассмотренный ранее рекурсивный метод вычисления факториала. Его последняя инструкция выглядит так:

```
return (n==0)?1:n*fact(n-1);
```

В задаче 6.1 после рекурсивного вызова функции `revers()` добавлена еще одна инструкция – печать на экран. Тот факт, что после рекурсивного вызова управление возвращается обратно в экземпляр метода, из которого вызов был сделан, позволил организовать вывод числовой последовательности в обратном порядке. Однако такой вид рекурсии уже нельзя отнести к хвостовой. Если хвостовую рекурсию можно реализовать итеративно, используя лишь фиксированное количество дополнительных переменных (не зависящее от глубины рекурсии), то метод `revers()` уже невозможно преобразовать в цикл без использования дополнительной памяти (например, массива).

6.3 Рекурсия с ветвлением

До сих пор мы рассматривали примеры рекурсивных методов, в которых каждый экземпляр метода делал не более одного рекурсивного вызова. Однако вызовов может быть и больше. Рекурсия, при которой метод может вызвать себя несколько раз, называется *ветвящейся* (или рекурсией *с ветвлением*). Примером ветвящейся рекурсии является рекурсивный метод вычисления чисел Фибоначчи. Рассмотрим его подробнее.

Задача 6.2 Напишите рекурсивный метод, который вычисляет n -ый член последовательности чисел Фибоначчи.

Объяснение: математически последовательность Фибоначчи можно определить, как:

$$fib(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ fib(n-1) + fib(n-2), & n > 1 \end{cases}$$

В методе будет два условия завершения рекурсии: при $n=0$ и $n=1$ и два рекурсивных вызова $fib(n-2)$ и $fib(n-1)$.

```
import java.util.Scanner;
public class FibR
{
    static long fib(int n) {
        return (n>1)?fib(n-1)+fib(n-2):(n==1)?1:0;
    }
    public static void main(String[] args)
    {
        Scanner scan=new Scanner(System.in);
        System.out.printf("Введите n:");
        int n = scan.nextInt();
        System.out.printf("fib(%d)= %d",n,fib(n));
    }
}
```

Результат:

```
Введите n:5
fib(5)= 5
```

Обратите внимание, что при вычислении $fib(5)$ в соответствии с этой программой понадобится найти $fib(4)$ и $fib(3)$. Определение $fib(4)$, в свою очередь, потребует вычисления $fib(3)$ и $fib(2)$, и так далее. Дерево вызовов метода $fib(5)$ приведено на рис.6.3. Внимательно изучение содержимого стека вызовов для этой задачи показывает, что для нахождения каждого следующего числа Фибоначчи требуется примерно *вдвое* большее время, чем для определения предыдущего. Поэтому для практического применения такая программа не пригодна.

6.4 Косвенная рекурсия

Косвенной или *взаимной* рекурсией называется организация вызовов нескольких методов по кругу (первый метод вызывает второй, второй – третий, ..., n -ый метод вызывает первый). Самым простым вариантом косвенной рекурсии можно считать случай, когда метод $a()$ вызывает метод $b()$, который вызывает метод $a()$. Рассмотрим рекурсивный способ вычисления значений функций синуса и косинуса в соответствии со следующими тождествами:

$$\sin x \equiv 2\sin(x/2)\cos(x/2), \quad \cos x \equiv \cos^2(x/2) - \sin^2(x/2).$$

Здесь функции синуса и косинуса вызывают одна другую и поэтому являются взаимно рекурсивными.

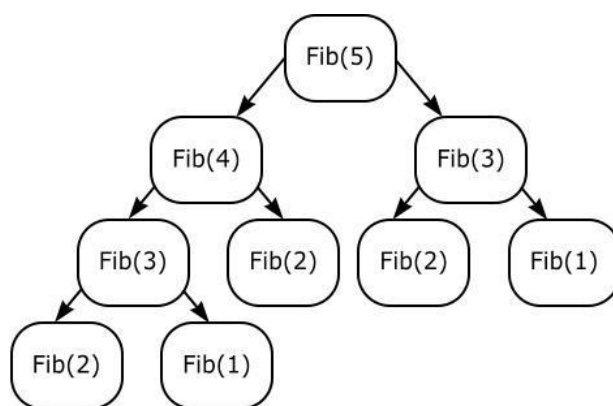


Рис.6.3 – Дерево стека вызовов рекурсивного метода `fib(5)`

Задача 6.3 Напишите рекурсивные методы вычисления синуса и косинуса.

Объяснение: условием завершения рекурсии будем считать равенство аргумента x нулю с точностью до величины $\text{eps}=0.001$. При возврате из самого глубокого рекурсивного вызова нужно будет вычислить нерекурсивно синус и косинус малого числа. Используем для этого первые два члена разложения тригонометрических функций в степенной ряд.

```

import java.util.Scanner;
public class Ex_6_3
{
    static double cos(double x){
        if (Math.abs(x)<10e-4)
            return x*(1-x*x/6);
        else
            return 2*sin(x/2)*cos(x/2);
    }

    static double sin(double x) {
        if (Math.abs(x)<10e-4)
            return (1-x*x/2);
        else
            return Math.pow(cos(x/2),2)-Math.pow(sin(x/2),2);
    }

    public static void main(String[] args)
    {
        Scanner scan=new Scanner(System.in);
        System.out.printf("Введите x:");
        double x = scan.nextDouble();
        System.out.printf("sin(x)= %5.5f\n",sin(x));
        System.out.printf("cos(x)= %5.5f",cos(x));
    }
}

```

```
}
```

Результат:

```
Введите x:1  
sin (x) = -0,54030  
cos (x) = -0,84147
```

6.5 Рекурсивный поиск в массивах

В главе 4 уже были изложены некоторые алгоритмы обработки одномерных массивов, однако их обсуждение можно считать не полным, без рассмотрения рекурсивных аналогов, в частности бинарного поиска и быстрой сортировки массивов.

Многие рекурсивные алгоритмы строятся по схеме «разделяй и властвуй», предполагающей наличие двух рекурсивных вызовов, каждый из которых работает приблизительно на половине входных данных. Это позволяет разделить задачу на независимые подзадачи того же типа. Часто использование подхода «разделяй и властвуй» обеспечивает более быстрые решения, чем итерационные алгоритмы.

В качестве примера рассмотрим задачу поиска максимального элемента в одномерном массиве. Итеративный вариант решения задачи был рассмотрен в главе 4. Рекурсивное решение типа «разделяй и властвуй» – еще один простой способ решения той же задачи.

Задача 6.4 *Напишите рекурсивный метод поиска максимального элемента в целочисленном массиве.*

Объяснение: рекурсивный метод будет проводить поиск максимального элемента в массиве `a[]` от индекса `left` (левая граница подмассива) до `right` (правая граница подмассива) включительно. В методе два рекурсивных вызова, каждый из которых работает на половине заданного участка массива. Задача сводится к нахождению максимального из двух элементов, и решается путем их сравнения. Если в подмассиве один элемент, то он возвращается методом как максимальный. Дерево вызовов рекурсивного метода `max()` для массива `a={5, 10, 12, 8}` представлено на рис. 6.4.

В программе используется новый способ вывода массива на экран. В методе `println()` применен метод `toString()` класса `java.util.Arrays`, который преобразует массив в строку.

Алгоритм работы рекурсивного метода:

1. Если индекс `left` равен индексу `right`, то завершить поиск и вернуть значение максимального элемента `a[left]`, иначе выполнить действия п.2-3.

2. Поделить анализируемый участок массива пополам. Вызвать метод поиска максимального значения для левого подмассива с границами left и (left+right)/2 и для правого подмассива с границами (left+right)/2+1 и right. Возвращаемые методами максимальные элементы для левого и правого участков исследуемого подмассива сохранить в дополнительных переменных temp1 и temp2 соответственно.

3. Вернуть максимальный из двух элементов temp1 и temp2.

```
public class Ex_6_4
{
    static int max(int[] a)
    {
        return max(a,0, a.length-1); }
    static int max (int[] a, int left, int right)
    {
        if (left==right) return a[left];
        else {
            int temp1= max(a,left, (left+right)/2);
            int temp2= max(a, (left+right)/2+1, right);
            return Math.max(temp1, temp2);}
    }
    public static void main(String[] args)
    {
        int[] a = {5,0,-10,9,15,100,-12,3};
        System.out.println(java.util.Arrays.toString(a));
        System.out.println("Max="+max(a));
    }
}
```

Результат:

```
[5, 0, -10, 9, 15, 100, -12, 3]
Max=100
```

Еще один алгоритм, построенный по принципу «разделяй и властвуй» – это *бинарный поиск* в упорядоченном массиве, который также называется методом половинного деления. Идея алгоритма заключается в следующем: если массив упорядоченный, то половину его элементов после сравнения ключа (искомого значения) со средним элементом можно отбросить. Если значение ключа меньше среднего элемента массива, то искомый элемент может находиться только в левой части массива, к которой применяется этот же метод бинарного поиска, если – больше, метод вызывается для правой части массива. Таким образом, в методе будет два рекурсивных вызова.

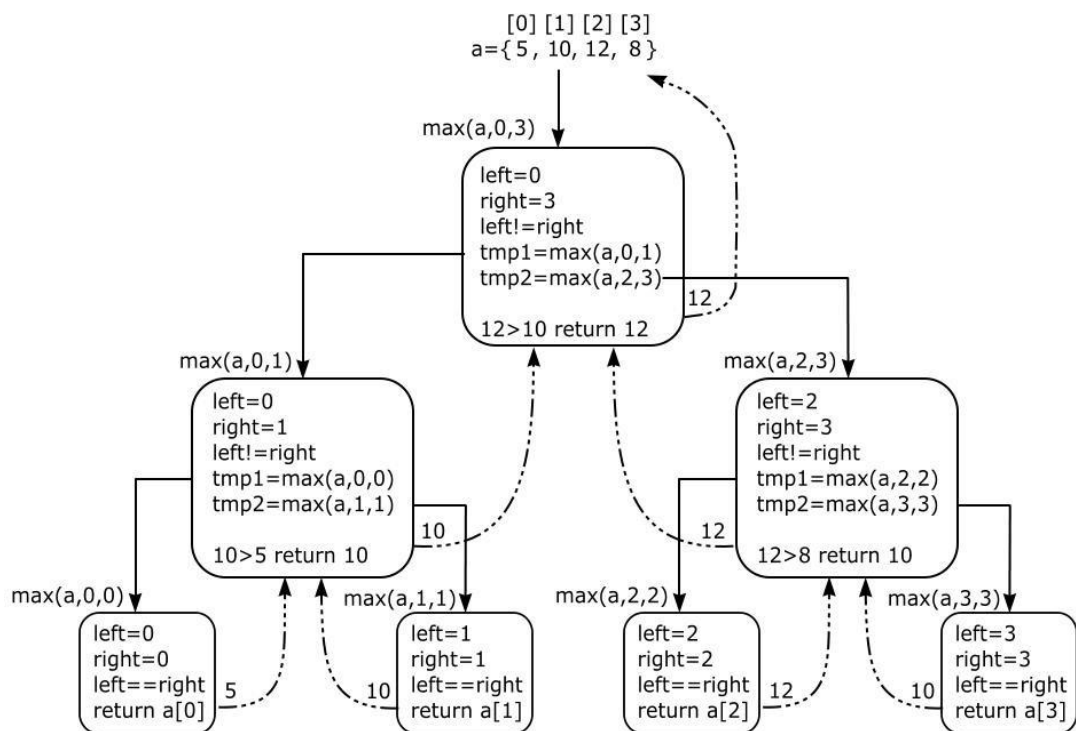


Рис. 6.4 – Дерево вызовов рекурсивного метода `max()`

Метод бинарного поиска работает значительно быстрее метода линейного поиска, рассмотренного в главе 4. Так, поиск в неотсортированных массивах занимает время, пропорциональное n , где n – количество элементов массива. При использовании метода бинарного поиска по отсортированным данным среднее время поиска будет пропорционально $\log_2 n$.

Задача 6.5 Напишите рекурсивный метод поиска в отсортированном целочисленном массиве.

Объяснение: в программе используются следующие переменные: массив `a[]`, `left` – индекс первого элемента (левая граница), `right` – индекс последнего элемента (правая граница), `middle` – индекс среднего элемента исследуемого участка массива, `key` – ключ поиска (искомый элемент). Метод возвращает индекс элемента, значение которого совпадает с ключом поиска. Если элемент не найден, возвращается -1. Условием завершения рекурсии является условие, при котором значение ключа совпадает со значением некоторого элемента массива, номер этого элемента возвращается. Если же индекс первого элемента исследуемого участка массива превышает индекс последнего элемента, то искомое значение в массиве отсутствует и возвращается -1.

Алгоритм работы рекурсивного метода:

1. Если индекс `left` больше индекса `right`, то завершить поиск и вернуть значение `-1`, иначе выполнить действия п.2-4.

2. Определить индекс среднего элемента:
`middle=(left+right)/2.`

3. Сравнить ключ поиска со средним значением. Если они совпадают, то поиск завершен и возвращается индекс среднего элемента – `middle`, иначе определить подмассив в котором следует продолжить поиск. Перейти к п.4.

4. Если ключ поиска меньше значения среднего элемента, вызвать рекурсивный метод бинарного поиска для левого подмассива с границами `left` и `middle-1`, иначе выполнить вызов метода бинарного поиска для правого подмассива с границами `middle+1` и `right`.

```
public class Ex_6_5
{
    static int binarySearch (int[] a, int left, int right, int
key)
    {
        if (left>right) return -1;
        else
        {
            int middle=(left+right)/2;
            if (key==a[middle]) return middle;
            else if(key<a[middle])
                return binarySearch(a,left,middle-1, key);
            else
                return binarySearch(a,middle+1, right, key);
        }
    }
    public static void main(String[] args)
    {
        int[] a = {-12, -10, 0, 3, 5, 9, 15, 100};
        System.out.println(java.util.Arrays.toString(a));
        System.out.printf("Search (9) index=%d",
binarySearch(a,0,a.length-1,9));
    }
}
```

Результат:

```
[-12, -10, 0, 3, 5, 9, 15, 100]
Search (9) index=5
```

6.6 Быстрая сортировка

Быстрая сортировка (англ. quicksort) – это надежный и эффективный алгоритм сортировки, который используется не только для учебных целей, но и широко применяется на практике. Быстрая сортировка

также называется сортировкой Хоара по имени автора алгоритма Чарльза Хоара. Для большинства массивов этот метод требует приблизительно $(n/6)\log n$ обменов элементов и $n\log n$ сравнений, то есть намного меньше, чем любой другой элементарный метод сортировки. Метод основан на подходе «разделяй и властвуй». Идея алгоритма достаточно проста. Из массива выбирается некоторый опорный элемент p , вокруг которого перемещаются все элементы. Причем элементы меньшие, либо равные p , перемещаются влево от него, а элементы большие, либо равные p – вправо. Далее метод быстрой сортировки рекурсивно запускается для каждой из частей массива. Алгоритм имеет наименьшее время сортировки, если в качестве опорного элемента выбирается средний по номеру элемент, так как в результате деления части массива имеют одинаковые размеры.

Задача 6.6 *Напишите рекурсивный метод быстрой сортировки целочисленного массива.*

Объяснение: введем два индекса i и j , в начале алгоритма они указывают, соответственно, на левый и правый конец последовательности. Начнем двигать указатель i с шагом 1 по массиву вперед, пока не будет найден элемент со значением большим или равным значению опорного элемента. Затем аналогичным образом двигаем указатель j от конца массива к началу, пока не будет найден элемент со значением меньшим или равным значению опорного элемента. Далее, если $i \leq j$, меняем найденные элементы местами и продолжаем двигать i и j . Алгоритм деления заканчиваем тогда, когда индекс i становится больше индекса j . После деления все значения до индекса i меньше или равны опорному элементу, а все значения после индекса j больше или равны опорному элементу. Далее рекурсивно вызываем метод быстрой сортировки отдельно для каждой из частей массива. Пример выполнения алгоритма быстрой сортировки массива {5, 0, -10, 9, 15, 100, -12, 3} иллюстрирует рис. 6.5. На рисунке показан только первый шаг рекурсии. На самом деле подмассивы {5, 0, -10, 3, -12} и {100, 15, 9} сортируются затем рекурсивно. Условием завершения рекурсивных вызовов является наличие одного элемента в каждой из частей массива.

```
public class QuickSort {
    static void quickSort (int[] a, int left, int right) {
        int i=left, j=right;
        int p=a[(left+right)/2]; //опорный элемент
        int tmp;
        // разделение
        do {
            while (a[i]<p) i++;
            while (a[j]>p) j--;
```

```

    if (i<=j) {
        tmp=a[i]; a[i]=a[j]; a[j]=tmp;
        i++; j--;
    }
    while (i<=j);

    // рекурсивные вызовы сортировки
    if (j>left) quickSort(a, left, j);
    if (i<right) quickSort(a, i, right);
}

public static void main(String[]args) {
    int[] a = {5,0,-10,9,15,100,-12,3};
    quickSort(a,0,a.length-1);
    System.out.println(java.util.Arrays.toString(a));
}

```

Результат:

[-12, -10, 0, 3, 5, 9, 15, 100]

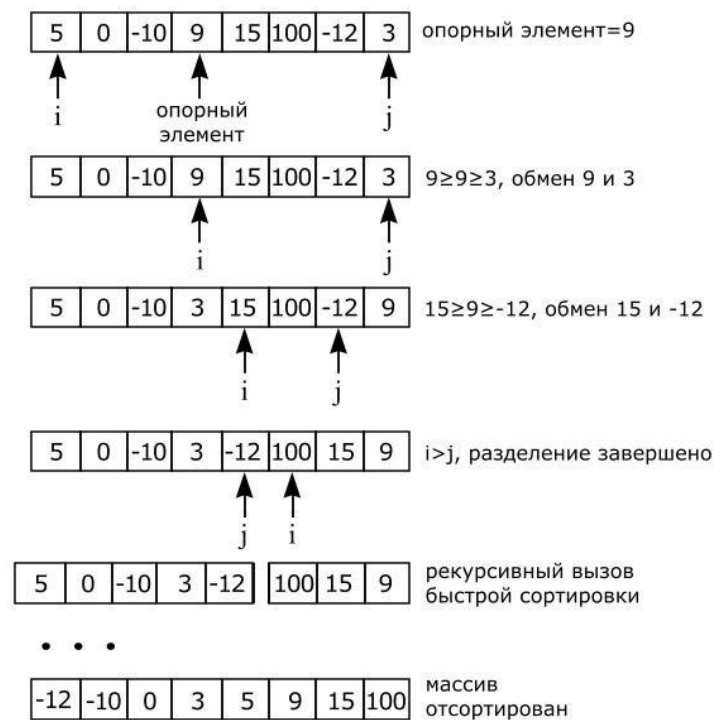


Рис.6.5 – Демонстрация выполнения алгоритма быстрой сортировки

6.7 Сортировка слиянием

Сортировка слиянием (англ. merge sort)²⁶ является простым, но эффективным алгоритмом сортировки, который также как и алгоритм быстрой сортировки построен по принципу «разделяй и властвуй», однако

²⁶ Метод сортировки слиянием был изобретен Джоном фон Нейман в 1945 году.

реализует его несколько по-другому. Массив делится пополам, левая и правая половина рекурсивно сортируется тем же методом слияния, затем отсортированные части объединяются в один отсортированный массив. В соответствии с алгоритмом деление продолжается до части массива, состоящего из одного элемента. Такой массив сразу является отсортированным, поэтому задача сводится к написанию метода, который выполняет слияния двух отсортированных частей массива в один.

Одним из способов слияния является выделение дополнительной памяти для вспомогательного буфера, равному по размеру общему количеству элементов в двух частях массива. Элементы последовательно перемещаются в этот буфер по одному за шаг. Причем на каждом шагу элементы двух частей сравниваются, и по результату сравнения в буфер помещается элемент либо с левого, либо с правого подмассива. Один из подмассивов во время слияния может закончиться раньше. В таком случае элементы другого подмассива, которые не были обработаны, дописываются в конец буфера, сохраняя имеющийся порядок. Результатом является упорядоченная последовательность, находящаяся в буфере.

Задача 6.7 *Напишите рекурсивный метод сортировки слиянием целочисленного массива.*

Объяснение: в программе используются следующие методы:

`mergeSort(int[] arr)` – выполняет сортировку массива `arr[]`, вызывая для этого метод `mergeSort(int[] arr, int low, int high, int[] buff)`;

`mergeSort(int[] arr, int low, int high, int[] buff)` – выполняет сортировку массива `arr[]` от индекса `low` до индекса `high`. В качестве аргумента в метод передается вспомогательный буфер `buff[]`. Метод осуществляет рекурсивное деление массива пополам и вызов метода слияния `merge()`.

`merge(int[] arr, int low, int high, int mid, int[] buff)` – метод выполняет слияние отсортированных частей массива: левой от индекса `low` до `mid` и правой от индекса `mid+1` до индекса `high`.

Пример выполнения алгоритма сортировки слиянием массива {5, 0, -10, 9, 15, 100, -12, 3} иллюстрирует рис. 6.5.

```
public class MergeSort {
    static void mergeSort(int[] arr) {
        mergeSort(arr, 0, arr.length - 1, new int[arr.length]);
    }
    static void mergeSort(int[] arr, int low, int high, int[] buff) {
        //Выполняем разделение
        if (low >= high) {
            return;
        }
    }
}
```

```

    }
    int mid = (low + high)/2;
    mergeSort(arr, low, mid, buff);
    mergeSort(arr, mid+1, high, buff);
    //Вызываем метод слияния
    merge(arr,low, high, mid, buff);
}

static void merge(int[]arr, int low, int high, int mid, int[]
buff)
{
    int left = low, right = mid + 1;
    //Выполняем слияние
    for (int i = low; i <= high; i++) {
        if (right>high||left<=mid && arr[left]<=arr[right])
        {
            buff[i] = arr[left++];
        } else {
            buff[i] = arr[right++];
        }
    }
    //Копируем буфер в массив arr
    for (int i = low; i <= high; i++) {
        arr[i] = buff[i];
    }
}

public static void main(String[]args) {
    int[] a = {5,0,-10,9,15,100,-12,3};
    mergeSort(a);
    System.out.println(java.util.Arrays.toString(a));
}
}

```

Результат:

```
[-12, -10, 0, 3, 5, 9, 15, 100]
```

Сортировка слиянием является одним из самых простых алгоритмов сортировки среди быстрых алгоритмов, который может быть эффективно использован для сортировки связанных списков. Недостаток алгоритма заключается в том, что он требует дополнительную память размером порядка n , и не гарантирует сохранение порядка элементов с одинаковыми значениями. Его временная сложность всегда пропорциональна $O(n \log n)$.

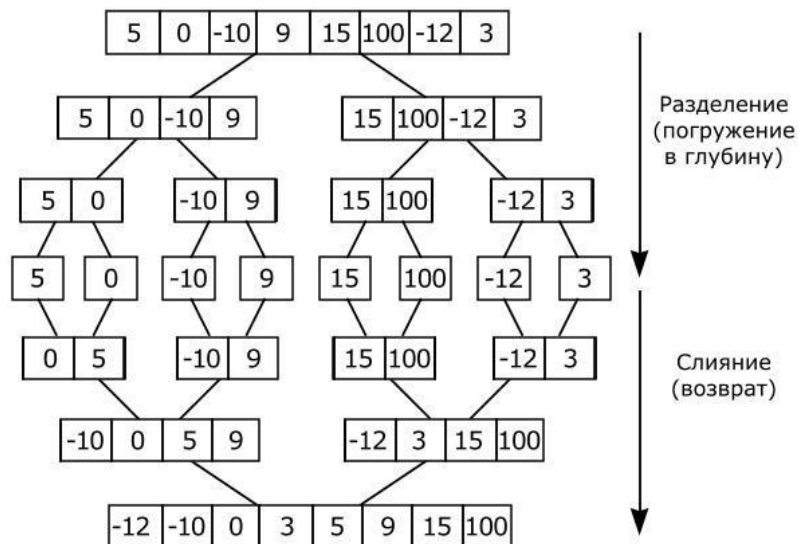


Рис.6.6 – Демонстрация выполнения алгоритма сортировки слиянием

6.8 Стандартные средства Java для работы с массивами

В классе `Arrays` из пакета `java.util` собраны статические методы, которые позволяют программисту облегчить работу, связанную с обработкой массивов. В частности, класс содержит методы, которые выполняют сравнение, заполнение, бинарный поиск и быструю сортировку массивов. Рассмотрим обзорно отдельные методы класса `Arrays`.

Метод `binarySearch()` использует бинарный поиск для нахождения заданного значения, применяется только к отсортированным массивам. Общая форма метода имеет вид:

```
static int binarySearch(type[] arr, type element)
```

Здесь `type`²⁷ – любой примитивный тип или тип `Object`. Метод возвращает индекс найденного элемента массива `arr[]`. Если элемент `element` не найден, то возвращается отрицательное число, абсолютное значение которого равно индексу, с которым элемент был бы вставлен в массив в заданном порядке. Если в массиве есть несколько одинаковых элементов, нет гарантии правильного поиска. Используемый алгоритм не был разработан в расчете на повторяющиеся элементы, хотя их присутствие не запрещается.

Методы `copyOf()` и `copyOfRange()` возвращают копию массива и имеют следующие общие формы:

```
static type[] copyOf(type[] arr, int len)
```

²⁷Метод `binarySearch()`, как и все другие рассматриваемые здесь статические методы класса `Arrays`, перегружен для всех примитивных типов и для типа `Object`.

```
static type[] copyOfRange(type[] arr, int from, int to)
```

Исходный массив задан параметром `arr[]`. В методе `copyOf()` в качестве параметра передается длина копии `len`. Если копия длиннее, чем источник, то для числовых массивов она дополняется нулями, для объектов – значениями `null`. Если копия короче, чем источник, она усекается. В методе `copyOfRange()` диапазон для копирования задается от индекса `from` включительно до индекса `to` исключительно.

Метод `equals()` сравнивает массивы и возвращает значение `true`, если два массива эквивалентны. В противном случае он возвращает значение `false`.

```
static boolean equals(type[] arr1, type[] arr2)
```

Здесь `arr1` и `arr2` – массивы, сравниваемые на эквивалентность. При сравнении массивов проверяются следующие условия: в массивах должно быть равное количество элементов и каждый элемент должен быть эквивалентен соответствующему элементу другого массива.

Метод `fill()` заполняет элементы массива заданным значением. Метод `fill()` имеет две версии: первая присваивает всем элементам массива `arr[]` значение `value`:

```
static void fill(type[] arr, type value)
```

Вторая версия метода `fill()` присваивает значение `value` подмножеству массива `arr[]` от индекса `from` включительно до индекса `to` исключительно.

```
static void fill(type[] arr, int from, int to, boolean value)
```

Метод `sort()` сортирует массив по возрастанию и имеет две версии. Первая версия сортирует весь массив. Общая форма имеет вид:

```
static void sort(type[] arr)
```

Здесь `arr[]` – это массив, подлежащий сортировке. Вторая версия метода `sort()` позволяет указать диапазон массива, который следует упорядочить:

```
static void sort(type[] arr, int from, int to)
```

Массив `arr[]` сортируется по возрастанию от индекса `from` включительно до `to` исключительно.

Обзор был бы не полным без описания метода копирования массивов `arraycopy()`, который использует сама исполняющая система Java. Метод находится в классе `System` в пакете `java.lang` и позволяет копировать массивы быстрее, чем это получается в случае поэлементного итеративного копирования. Метод перегружен для всех примитивных типов и типа `Object`. Общая форма метода имеет вид:

```
static void arraycopy(type[] src, int src_ind, type[] dest, int dest_ind, int count)
```

Из массива, на который указывает ссылка `src`, копируется `count` элементов, начиная с элемента с индексом `src_ind`, в массив, на который указывает ссылка `dest`, начиная с его элемента с индексом `dest_ind`. Все индексы должны быть заданы так, чтобы элементы лежали в массивах, типы массивов должны быть совместимы, а примитивные типы обязаны полностью совпадать. Ссылки на массивы не должны быть равны `null`.

Рассмотрим пример демонстрирующий применение некоторых из представленных выше методов.

Задача 6.8 Дан двумерный массив целых чисел, состоящий из $n \times m$ элементов. Все строки массива, которые упорядочены по возрастанию, заменить нулями.

```
import java.util.*;
class Ex_6_8{
    //метод инициализации массива
    static void fill(int [][] a){
        for(int i=0;i<a.length;i++)
            for(int j=0;j<a[i].length;j++)
                if (i%3==0) a[i][j]=j;
                else a[i][j]=(int) (Math.random()*10)-5;
    }
    //метод вывода массива на печать
    static void print(int[][] a){
        for(int i=0;i<a.length;i++)
            System.out.println(Arrays.toString(a[i]));
    }
    //логический метод определяет отсортирован ли массив
    static boolean is_sort(int[] t){
        int[] t1=new int[t.length];
        System.arraycopy(t,0,t1,0,t.length);
        Arrays.sort(t1);
        return (Arrays.equals(t,t1));
    }
    public static void main(String[] args) {
        int n=5, m=5;
        int[][] arr=new int[n][m];
        fill(arr);
        System.out.println("Исходная матрица");
        print(arr);
        for(int i=0;i<arr.length;i++){
            if (is_sort(arr[i])) Arrays.fill(arr[i],0);
        }
        System.out.println("Новая матрица");
        print(arr); } }
```

Результат:

Исходная матрица

[0, 1, 2, 3, 4]
[-5, -1, -4, 0, 0]
[4, -1, -3, -5, 0]
[0, 1, 2, 3, 4]
[-3, -5, 4, 2, 3]

Новая матрица

[0, 0, 0, 0, 0]
[-5, -1, -4, 0, 0]
[4, -1, -3, -5, 0]
[0, 0, 0, 0, 0]
[-3, -5, 4, 2, 3]

Упражнения к главе 6

1. Опишите рекурсивный метод для выполнения следующего задания.
Дано вещественное число x и целое число n . Определить x^n .
Степенную функцию вычислять по формуле:

$$x^n = \begin{cases} 1, & n = 0, \\ 1/x^{|n|}, & n < 0, \\ x \cdot x^{n-1}, & n > 0. \end{cases}$$

2. Опишите рекурсивный метод, который для двух заданных натуральных чисел находит наибольший общий делитель по алгоритму Евклида.
3. Опишите рекурсивный метод умножения двух натуральных чисел, используя рекуррентное соотношение:

$$a \cdot b = \begin{cases} b, & a = 1, \\ (a-1) \cdot b + b, & a > 1. \end{cases}$$

4. Вычислите, используя рекурсию:

$$\sqrt{1 + (n+1)\sqrt{1 + (n+2)\sqrt{1 + (n+3)\sqrt{1 + \dots}}}}$$

Ответ: $n+2$

5. Подсчитайте, сколько раз потребуется повторно вычислить четвёртый элементы последовательности Фибоначчи для вычисления пятнадцатого элемента.
6. Напишите рекурсивный метод сложения двух чисел, используя следующее рекуррентное соотношение:

$$a + b = \begin{cases} a, & \text{if } b = 0, \\ (a + 1) + (b - 1), & \text{if } b > 0, \\ (a - 1) + (b + 1), & \text{if } b < 0. \end{cases}$$

7. Используя рекурсию, найдите n -ый член последовательности, которая определяется следующим рекуррентным соотношением: $a_1 = 1, a_2 = 2, a_{n+1} = 2a_n + a_{n-1}$.

8. Вычислите функцию Аккермана по ее рекурсивному определению:

$$A(n, m) = \begin{cases} m + 1, & n = 0, \\ A(n - 1, 1), & n \neq 0, m = 0, \\ A(n - 1, A(n, m - 1)), & n > 0, m > 0. \end{cases}$$

9. Опишите рекурсивный метод, который методом деления отрезка пополам находит с заданной точностью eps корень уравнения $f(x) = 0$ на отрезке $[a, b]$, $eps > 0, a < b, f(a) \cdot f(b) < 0$. Подсказка: между точками a и b существует корень, если $f(a)$ и $f(b)$ имеют разные знаки.

10. Напишите рекурсивный метод нахождения суммы первых n членов арифметической прогрессии $1^3 + 3^3 + 5^3 + \dots + (2n - 1)^3$.

11. Вычислите количество сочетаний по рекуррентной формуле биномиальных коэффициентов:

$$C_n^k = \begin{cases} 0, & k > n, \\ 1, & k = 0 \text{ или } k = n, \\ C_{n-1}^k + C_{n-1}^{k-1}, & 0 < k < n. \end{cases}$$

12. Напишите рекурсивный метод вывода цифр целого положительного числа n в обратном порядке.

13. Опишите рекурсивный метод нахождения суммы цифр любого натурального числа.

14. Задано число n . Сложить все цифры числа n , затем все цифры найденной суммы и повторить эти действия до тех пор, пока не получим цифру, называемую цифровым корнем числа. Напишите рекурсивный метод вычисления цифрового корня числа n .

15. Напишите рекурсивный метод перевода натурального числа из десятичной системы счисления в двоичную.

7 СТРОКИ И ТЕКСТОВЫЙ ВВОД-ВЫВОД

Строка – это последовательность символов. Символы в строках хранятся в кодировке Unicode, в которой каждый символ занимает два байта. Тип каждого символа `char`.

Во многих языках программирования строки представляют собой массивы символов, но в Java строки являются объектами. Они могут быть экземплярами класса `String` или классов `StringBuffer` и `StringBuilder`.

В объектах класса `String` хранятся строки-константы неизменной длины и содержания. Это значительно ускоряет обработку строк и позволяет экономить память, разделяя строку между объектами, использующими ее. Когда говорят, что строки в объектах класса `String` неизменяемы, имеют в виду, что содержимое экземпляра строки не может быть изменено после ее создания. Однако переменная, объявленная как `String`, в любой момент может быть переопределена так, чтобы указывать на другой объект класса `String`.

Длину строк, хранящихся в объектах класса `StringBuffer` и `StringBuilder`, можно менять, вставляя и добавляя строки и символы, удаляя подстроки или сцепляя несколько строк в одну строку.

Классы `String`, `StringBuffer` и `StringBuilder` определены в пакете `java.lang` и доступны автоматически.

7.1 Класс `String`

7.1.1 Конструкторы класса `String`

Класс `String` имеет 13 конструкторов и более чем 40 методов для манипулирования строками.

Строку можно создать из строкового литерала или массива символов. Чтобы создать строку из строкового литерала следует использовать следующий синтаксис:

```
String newString = new String(stringLiteral);
```

Аргумент `stringLiteral` – это последовательность символов, заключенная в двойные кавычки. Следующий оператор создает строку `message` из строкового литерала `"Welcome to Java"`:

```
String message = new String("Welcome to Java");
```

Java преобразует строковый литерал в объект типа `String`. Создать строку можно и следующим образом:

```
String message = "Welcome to Java";
```

Создать строку можно из массива символов. Следующая инструкция создает строку `"Good Day"`:

```
char[] charArray = {'G', 'o', 'o', 'd', ' ', 'D', 'a', 'y'};
```

```
String message = new String(charArray);
```

Некоторые другие часто используемые конструкторы класса `String` приведены в примере 7.1.

Пример 7.1 Конструкторы класса *String*.

Объяснение: Первая строка в примере создается конструктором класса `String` без параметров и назначает своей ссылкой переменную `s1`. Созданный при этом новый объект не содержит символов (это пустая строка, которая записывается в виде "" и имеет длину 0). Объект с ссылкой `s2` создается конструктором класса `String`, принимающим в качестве аргумента другой строковый объект. Новый объект будет содержать ту же последовательность символов, что и строковый объект, переданный в конструктор в качестве аргумента. Объект `s3` создается из массива символов `charArray`. А объект `s4`, создается конструктором, который принимает в качестве аргументов массив символов `charArray` и два целых числа. Второй аргумент определяет начальную позицию (`offset`), начиная с которой будут выбираться символы из массива в создаваемую строку. Третий аргумент указывает количество символов (`count`). В случае если аргументы `offset` и `count` имеют недопустимые значения, выходящие за пределы массива, будет брошено исключение `StringIndexOutOfBoundsException`.

```
public class StringConstructors
{
    public static void main( String[] args )
    {
        char[] charArray = {'b','i','r','t','h','','d','a','y'};
        String s = new String( "hello" );
        // use String constructors
        String s1 = new String();
        String s2 = new String(s);
        String s3 = new String(charArray);
        String s4 = new String(charArray, 6, 3);
        System.out.printf("s1 = %s\ns2 = %s\ns3 = %s\ns4 = %s\n",
                           s1, s2, s3, s4 );
    }
}
```

Результат:

```
s1 =
s2 = hello
s3 = birth day
s4 = day
```

Будьте внимательны! Переменная типа `String` содержит ссылку на объект-строку, которая хранит значение строки. Строго говоря, переменная `String`, объект `String`, и строковое значение – разные. Большую часть времени различия между ними будем игнорировать. Для простоты термин строка будет часто использоваться для обозначения переменной `String`, объекта `String`, и самого значения строки.

7.1.2 Интернирование строк

Объект `String` является неизменным, его содержание не может быть изменено. Изменит ли следующий код содержимое строки?

```
String s = "Java";  
s = "HTML";
```

Ответ: нет. Первый оператор создает объект `String` с содержанием "Java" и ссылкой `s`. Второй оператор создает новый объект `String` с содержанием "HTML" и присвоит его ссылке `s`. Первый объект `String` все еще существует после присваивания, но к нему больше нельзя получить доступ, потому что переменная `s` теперь указывает на новый объект, как показано на рисунке 7.1.

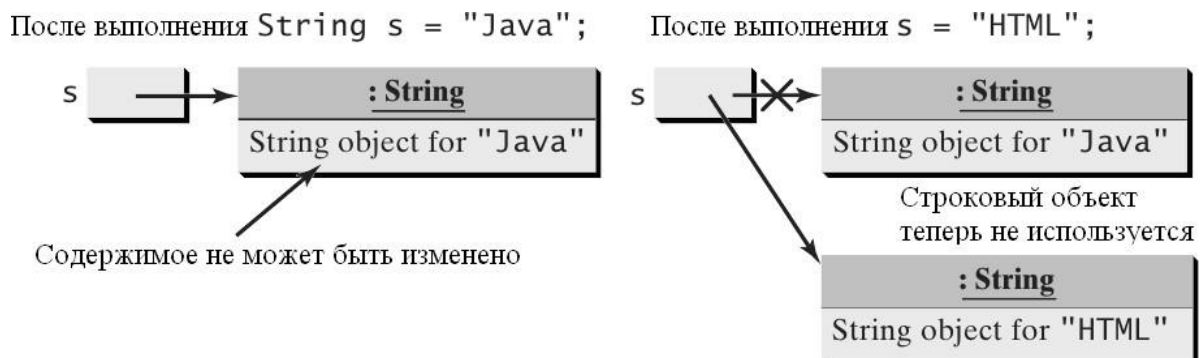


Рисунок 7.1 – Строки `String` являются неизменными, однажды создав их содержимое нельзя изменить

Поскольку строки являются неизменными и часто применяются в программировании, JVM использует уникальный экземпляр для строковых литералов с одинаковой последовательностью символов в целях повышения эффективности и экономии памяти. Если строки идентичны, две ссылки могут указывать на один объект, что позволяет более экономно использовать память и быстрее сравнивать строки – этот механизм называется *интернированием строк*. Например, следующие операторы:

```
String s1 = "Welcome to Java";  
String s2 = new String("Welcome to Java");  
String s3 = "Welcome to Java";  
System.out.println("s1==s2 is" + (s1==s2));
```

```
System.out.println("s1==s3 is" + (s1==s3));
напечатают:
s1==s2 is false
s1==s3 is true
```

Ссылки `s1` и `s3` указывают на один и тот же объект – интернированную строку "Welcome to Java", поэтому сравнение `s1 == s3` – это истина. Тем не менее, операция сравнения `s1 == s2` является ложной, так как `s1` и `s2` два различных объекта `String`, не смотря на то, что у них одинаковое содержимое (рис.7.2).

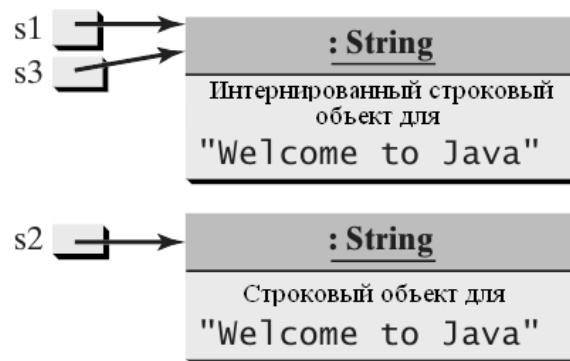


Рисунок 7.2 – Пример интернирования строк

7.1.3 Нахождение длины строки и выбор символов из строки

Методы класса `String`: `length()`, `charAt()` и `getChars()` возвращают длину строки, символ строки из указанной позиции и набор символов из строки в виде массива символов, соответственно.

Пример 7.2 Применение методов `length()`, `charAt()` и `getChars()` класса `String`.

```
public class StringExample
{
    public static void main( String[] args )
    {
        String s1 = "hello there";
        char[] charArray = new char[5];
        System.out.printf( "s1: %s",s1);

        // тестируем метод length()
        System.out.printf( "\nLength of s1: %d", s1.length());

        // перебор символов строки s1 с помощью
        // метода charAt() в обратном порядке
        System.out.print( "\nThe string reversed is: " );
        for ( int count = s1.length() - 1; count >= 0; count-- )
```

```
System.out.printf("%c",s1.charAt(count));

// копируем символы из строки в массив charArray
s1.getChars( 0, 5, charArray, 0 );
System.out.print( "\nThe character array is: " );
for ( char character : charArray )
System.out.print( character );
System.out.println();
}}
```

Результат:

```
s1: hello there
Length of s1: 11
The string reversed is: ereht olleh
The character array is: hello
```

В коде примера 7.2 метод `length()` используется для определения количества символов в строке `s1`. Как и массивы, строки знают свою длину. Однако, в отличие от массивов, доступ к длине строки можно получить только через метод `length()` класса `String`.

Метод `charAt()` возвращает из строки символ, расположенный в заданной позиции. Метод `charAt()` принимает целочисленный аргумент, который используется в качестве индекса и возвращает символ из этой позиции. Как и в массивах, первый элемент строки имеет нулевой индекс. Если индекс, переданный в качестве аргумента в метод, отрицателен или больше чем длина строки, возникает исключительная ситуация. В программе с помощью метода `charAt()` организован вывод символов строки `s1` в обратном порядке.

Метод `getChars()` используется в программе для копирования символов строки в массив символов. Первый аргумент метода – это индекс, начиная с которого будет выполняться копирование символов. Вторым аргументом – это индекс, до которого будет выполняться копирование символов (исключая символ в данной позиции). Третьим аргументом является массив, в который символы должны быть скопированы. Последним аргументом является отправной индекс, начиная с которого скопированные символы помещаются в целевой массив.

Задача 7.1. *Определите, является ли введенная строка палиндромом.*

Объяснение: строка является палиндромом, если она одинаково читается в обе стороны. Слова "топот", "шалаш" и "ротатор", например, являются палиндромами.

Одно из решений задачи состоит в проверке, является ли первый символ в строке таким же, как последний символ. Если это так, проверяется второй и предпоследний символ. Далее этот процесс

продолжается до тех пор, пока не будет найдено несоответствие или все символы в строке будут сравнены, за исключением среднего символа, если строка имеет нечетное количество символов.

Для реализации этой идеи, в программе используются две переменные `low` и `high`, равные индексам двух символов в начале и в конце строки `s` соответственно. Изначально переменная `low` равна 0, и `high` равна `s.length() - 1`. Если символы с указанными индексами равны, то `low` увеличивается на 1, а `high` соответственно уменьшается на 1. Этот процесс продолжается до тех пор, пока (`low < high`) или до момента, когда не будет найдено соответствие.

```
import java.util.Scanner;
public class CheckPalindrome {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in, "UTF-8");
        System.out.print("Введите строку: ");
        String s = input.nextLine();
        if (isPalindrome(s))
            System.out.println(s + " является палиндромом");
        else
            System.out.println(s + " не является палиндромом");
    }

    public static boolean isPalindrome(String s) {
        int low = 0;
        int high = s.length() - 1;
        while (low < high) {
            if(s.charAt(low) != s.charAt(high))
                return false; // не палиндром
            low++;
            high--;
        }
        return true; // строка является палиндромом
    }
}
```

Результат:

```
Введите строку: мама
мама не является палиндромом
Введите строку: шалаш
шалаш является палиндромом
```

7.1.4 Сравнение строк

Класс `String` предоставляет методы для сравнения строк, которые приведены в табл.7.1.

Таблица 7.1 – Методы класса String для сравнения строк

Метод	Описание
<code>boolean equals(Object s1)</code>	Возвращает true если строка равна строке s1
<code>boolean equalsIgnoreCase(String s1)</code>	Возвращает true если данная строка равна строке s1 без учета регистра букв
<code>int compareTo(String s1)</code>	Возвращает целое число больше 0, равное 0 или меньше 0 в зависимости от того является ли данная строка больше, равной или меньше чем s1.
<code>int compareToIgnoreCase(String s1)</code>	То же, что и метод <code>compareTo()</code> за исключением того, что при сравнении не учитываются регистры букв
<code>boolean regionMatches(int index, String s1, int s1Index, int len)</code>	Возвращает true если указанная подстрока точно соответствует указанной подстроке строки s1
<code>boolean regionMatches(boolean ignoreCase, int index, String s1, int s1Index, int len)</code>	То же, что в предыдущем методе исключением того, что можно указать параметр true – игнорировать регистр букв
<code>boolean startsWith(String prefix)</code>	Возвращает true если строка начинается с указанного префикса
<code>boolean endsWith(String suffix)</code>	Возвращает true если строка заканчивается указанным суффиксом

Как сравнить содержимое двух строк? Можно попытаться использовать оператор сравнения `==`, а именно:

```
if (string1==string2)
    System.out.println("string1 и string2 - один и тот же объект");
```

```
else
    System.out.println("string1 и string2 - разные объекты");
```

Операция сравнения `==` сопоставляет только ссылки на строки. Она выясняет, указывают ли ссылки на одну и ту же строку. Например, для строк

```
String s1 = "Какая-то строка";
String s2 = "Другая строка";
```

сравнение `s1 == s2` дает в результате `false`.

Значение `true` получится, только если обе ссылки указывают на одну и ту же строку, например, после присваивания `s1 = s2`.

Интересно, что если мы определим s2 так:

```
String s2 = "Какая-то строка";
```


то сравнение `s1 == s2` даст в результате `true`, потому что компилятор создаст только один интернированный экземпляр константы "Какая-то строка" и направит на него все ссылки.

Если нужно сравнивать не ссылки, а содержимое строк, то для этого есть несколько методов.

Логический метод `equals (Object obj)`, переопределенный из класса `Object`, возвращает `true`, если аргумент `obj` не равен `null`, является объектом класса `String`, и строка, содержащаяся в нем, полностью идентична данной строке вплоть до совпадения регистра букв. В остальных случаях возвращается значение `false`. Для сравнения содержимого двух строк следовало бы использовать следующий код:

```
if (string1.equals(string2))
    System.out.println("string1 и string2 имеют одинаковое
содержимое");
```

```
else
    System.out.println("string1 и string2 не равны");
```

Пример сравнения строк с помощью метода `equals()` приведен ниже:

```
String s1 = new String("Welcome to Java");
String s2 = "Welcome to Java";
String s3 = "Welcome to C++";
System.out.println(s1.equals(s2)); // true
System.out.println(s1.equals(s3)); // false
```

Метод `compareTo()` также может быть использован для сравнения двух строк. Например, так: `s1.compareTo (s2)`;

Метод возвращает значение 0, если `s1` равно `s2`, значение меньше 0, если `s1` *лексикографически* (то есть, с точки зрения упорядочения Unicode) меньше, чем `s2`, а значение больше 0, если `s1` лексикографически больше `s2`.

Фактическое значение, возвращаемое методом `compareTo()` зависит от смещения первых двух различных символов `s1` и `s2` слева направо. Например, предположим, `s1` равно `abc`, а `s2` равно `abg`, то `s1.compareTo (s2)` вернет -4. Сначала будут сравниваться первые два символа (`a` и `a`) у `s1` и `s2`. Так как они равны, сравнивается вторая пара символов (`b` и `b`). Они также равны, далее сравниваются третьи два символа (`c` и `g`). Поскольку символ `c` на 4 меньше, чем `g`, то метод сравнения вернет -4.

Запомните! Метод `equas()` возвращает `true`, если две строки равны и `false`, если они не равны. Метод `compareTo()` возвращает 0, целое положительное число или целое отрицательное число, в зависимости от того равна, больше или меньше сравниваемая строка, по сравнению со второй строкой.

Класс `String` также имеет методы `equalsIgnoreCase()`, `compareToIgnoreCase()` и `regionMatches()` для сравнения строк.

Методы `equalsIgnoreCase()` и `compareToIgnoreCase()` игнорируют регистр букв при сравнении двух строк.

Метод `regionMatches()` сравнивает подстроки двух строк на равенство:

```
regionMatches(int index, String s1, int s1Index, int len)
```

здесь `index` – индекс начала подстроки данной строки, `s1Index` – индекс начала подстроки строки `s1`. Метод различает символы, записанные в разных регистрах. Если надо сравнивать подстроки без учета регистров букв, то используйте логический метод:

```
regionMatches(boolean ignoreCase, int index, String s1, int s1Index, int len)
```

Если первый параметр `ignoreCase` равен `true`, то регистр букв при сравнении подстрок не учитывается, если `false` – учитывается.

Способы применения методов сравнения строк приведены в коде примера 7.3.

Пример 7.3 *Применение методов сравнения строк.*

```
public class StringCompare
{
    public static void main( String[] args )
    {
        String s1 = new String( "hello" );
        String s2 = "goodbye";
        String s3 = "Happy Birthday";
        String s4 = "happy birthday";
        System.out.printf("s1 = %s\n s2 = %s\n s3 = %s\n s4 = %s\n\n", s1, s2, s3, s4 );

        //тест для метода equals
        if(s1.equals("hello"))
            System.out.println( "s1 равна \"hello\"" );
        else
            System.out.println( "s1 не равна \"hello\"" );

        // тест для операции сравнения ==
        if (s1=="hello")
            System.out.println( "s1 один и тот же объект, что и \"hello\"" );
        else
            System.out.println( "s1 не один и тот же объект, что и \"hello\"" );

        // тест для метода equals (ignore case)
        if(s3.equalsIgnoreCase(s4))
            System.out.printf( "%s равна %s без учета регистра букв\n", s3, s4 );
    }
}
```

```

else
    System.out.println( "s3 не равна s4" );

// тест compareTo
System.out.printf("\ns1.compareTo(s2)=%d",
s1.compareTo(s2));
System.out.printf("\ns2.compareTo(s1)=%d",
s2.compareTo(s1));
System.out.printf("\ns1.compareTo(s1)=%d",
s1.compareTo(s1));
System.out.printf("\ns3.compareTo(s4)=%d",
s3.compareTo(s4));
System.out.printf("\ns4.compareTo(s3)=%d\n\n",
s4.compareTo(s3));

// тест regionMatches
if(s3.regionMatches(0, s4, 0, 5))
    System.out.println( "Первые 5 символов s3 и s4 равны");
else
    System.out.println("Первые 5 символов s3 и s4 не равны");

// тест regionMatches (ignore case)
if(s3.regionMatches(true, 0, s4, 0, 5))
    System.out.println("Первые 5 символов s3 и s4 равны
без учета регистра");
else
    System.out.println("Первые 5 символов s3 и s4
не равны");
}}

```

Результат:

```

s1 = hello
s2 = goodbye
s3 = Happy Birthday
s4 = happy birthday

```

```

s1 равна "hello"
s1 не один и тот же объект, что и "hello"
Happy Birthday равна happy birthday без учета регистра букв

```

```

s1.compareTo(s2) = 1
s2.compareTo(s1) = -1
s1.compareTo(s1) = 0
s3.compareTo(s4) = -32
s4.compareTo(s3) = 32

```

```

Первые 5 символов s3 и s4 не равны
Первые 5 символов s3 и s4 равны без учета регистра

```

Для того чтобы проверить, не начинается ли данная строка с подстроки-префикса `prefix`, используйте логический метод `startsWith(String prefix)`, возвращающий `true`, если строка начинается с данного префикса, или совпадает с ней, или префикс пуст.

Можно проверить и появление подстроки `sub` в данной строке, начиная с некоторого индекса `ind` логическим методом `startsWith(String sub, int ind)`. Если индекс `ind` отрицателен или больше длины строки, возвращается `false`.

Для того чтобы проверить, не заканчивается ли данная строка суффиксом `suffix`, используйте логический метод `endsWith(String suffix)`. Учтите, что он возвращает `true`, если суффикс `suffix` совпадает со всей строкой или суффикс `suffix` пуст.

Например, таким образом, `if (fileName.endsWith(". java"))`, можно отследит имена файлов с исходными текстами Java. Другие варианты применения методов строк `startsWith()` и `endsWith()` приведены в коде примера 7.4.

Пример 7.4 *Применение методов для сравнения строк `startsWith()` и `endsWith()`.*

```
public class StringStartEnd
{
    public static void main( String[] args )
    {
        String[] strings = {"started","starting","ended","ending"};

        // тест startsWith
        for (String string : strings)
        {
            if(string.startsWith("st"))
                System.out.printf("\n%s\n" начинается с \"st\"\n", string);
        }
        System.out.println();

        // тест startsWith с позиции 2
        for (String string : strings)
        {
            if(string.startsWith("art", 2))
                System.out.printf("\n%s\n" начинается с \"art\" в позиции 2\n", string);
        }
        System.out.println();

        // тест endsWith
        for (String string : strings)
        {
```

```

if(string.endsWith("ed"))
System.out.printf("\"%s\" заканчивается на \"ed\\n\",
string);
}
}}

```

Результат:

```

"started" начинается с "st"
"starting" начинается с "st"

"started" начинается с "art" в позиции 2
"starting" начинается с "art" в позиции 2

"started" заканчивается на "ed"
"ended" заканчивается на "ed"

```

7.1.5 Поиск символов и подстроки в строке

Часто бывает полезно найти первое вхождение символа или набора символов. Например, при создании собственного текстового редактора, такая функциональность может обеспечить поиск по документам. Основные методы класса `String` для организации поиска символов и подстроки в строке приведены в табл.7.2.

Таблица 7.2 – Методы класса `String` для поиска символов и подстроки в строке

Метод	Описание
<code>int indexOf (char ch)</code>	Возвращает индекс первого вхождения символа <code>ch</code> в строку или -1, если символ не найден
<code>int indexOf (char ch, int fromIndex)</code>	Возвращает индекс первого вхождения символа <code>ch</code> после индекса <code>fromIndex</code> в строку или -1, если символ не найден
<code>int indexOf (String s)</code>	Возвращает индекс первого вхождения подстроки <code>s</code> в строку или -1, если подстрока не найдена
<code>int indexOf (String s, int fromIndex)</code>	Возвращает индекс первого вхождения подстроки <code>s</code> в строку после индекса <code>fromIndex</code> или -1, если подстрока не найдена
<code>int lastIndexOf (char ch)</code>	Возвращает индекс последнего вхождения символа <code>ch</code> в строку или -1, если символ не найден

<code>int lastIndexOf (char ch, int fromIndex)</code>	Возвращает индекс последнего вхождения символа <code>ch</code> до индекса <code>fromIndex</code> в строку или <code>-1</code> , если символ не найден
<code>int lastIndexOf (String s)</code>	Возвращает индекс последнего вхождения подстроки <code>s</code> в строку или <code>-1</code> , если подстрока не найдена
<code>int lastIndexOf (String s, int fromIndex)</code>	Возвращает индекс последнего вхождения подстроки <code>s</code> в строку до индекса <code>fromIndex</code> или <code>-1</code> , если подстрока не найдена

Ниже приведены варианты применения методов из табл.7.2,

```
"Welcome to Java".indexOf('W') returns 0.
"Welcome to Java".indexOf('o') returns 4.
"Welcome to Java".indexOf('o', 5) returns 9.
"Welcome to Java".indexOf("come") returns 3.
"Welcome to Java".indexOf("Java", 5) returns 11.
"Welcome to Java".indexOf("java", 5) returns -1.
"Welcome to Java".lastIndexOf('W') returns 0.
"Welcome to Java".lastIndexOf('o') returns 9.
"Welcome to Java".lastIndexOf('o', 5) returns 4.
"Welcome to Java".lastIndexOf("come") returns 3.
"Welcome to Java".lastIndexOf("Java", 5) returns -1.
"Welcome to Java".lastIndexOf("Java") returns 11.
```

Другие варианты применения методов поиска символов и подстроки в строке представлены в коде примера 7.5.

Пример 7.5 *Применение методов для поиска символов и подстроки в строке `indexOf()` и `lastIndexOf()`.*

```
public class StringIndexMethods
{
    public static void main( String[] args )
    {
        String letters = "abcdefghijklmabcdefghijklm";

        // тестирование indexOf для поиска символов в строке
        System.out.printf("'c' найден с индексом %d\n",
            letters.indexOf('c'));
        System.out.printf("'a' найден с индексом %d\n",
            letters.indexOf('a',1));
        System.out.printf("' '$' найден с индексом %d\n\n",
            letters.indexOf('$'));

        // тестирование lastIndexOf для поиска символов в строке
        System.out.printf("Последний 'c' найден с индексом %d\n",
```

```

        letters.lastIndexOf('c'));
System.out.printf("Последний 'a' найден с индексом %d\n",
        letters.lastIndexOf('a', 25));
System.out.printf("Последний '$' найден с индексом
        %d\n\n", letters.lastIndexOf('$'));

// тестирование indexOf для поиска подстроки в строке
System.out.printf("Подстрока \"def\" найдена с индекса
        %d\n", letters.indexOf("def"));
System.out.printf("Подстрока \"def\" найдена с индекса
        %d\n", letters.indexOf("def", 7));
System.out.printf("Подстрока \"hello\" найдена с индекса
        %d\n\n", letters.indexOf("hello"));

// тестирование lastIndexOf для поиска подстроки в строке
System.out.printf("Последняя подстрока \"def\" найдена с
        индекса %d\n", letters.lastIndexOf("def"));
System.out.printf("Последняя подстрока \"def\" найдена с
        индекса %d\n", letters.lastIndexOf("def", 25));
System.out.printf("Последняя подстрока \"hello\" найдена с
        индекса %d\n", letters.lastIndexOf("hello"));
}}

```

Результат:

```

'c' найден с индексом 2
'a' найден с индексом 13
'$' найден с индексом -1

```

```

Последний 'c' найден с индексом 15
Последний 'a' найден с индексом 13
Последний '$' найден с индексом -1

```

```

Подстрока "def" найдена с индекса 3
Подстрока "def" найдена с индекса 16
Подстрока "hello" найдена с индекса -1

```

```

Последняя подстрока "def" найдена с индекса 16
Последняя подстрока "def" найдена с индекса 16
Последняя подстрока "hello" найдена с индекса -1

```

Помните! Поиск всегда ведется с учетом регистра букв.

7.1.6 Извлечение подстроки из строки

Для извлечения подстроки из строки в классе `String` имеются методы `substring()`, подробно описанные в табл. 7.3.

Таблица 7.3 – Методы класса String для извлечения подстроки из строки

Метод	Описание
String substring (int beginIndex)	Возвращает подстроку из строки от индекса beginIndex включительно и до конца строки, как показано на рис. 7.3. Если индекс beginIndex отрицательный или больше длины строки, то возникает исключительная ситуация.
String substring (int beginIndex, int endIndex)	Возвращает подстроку из строки от символа с индексом beginIndex включительно до символа с индексом endIndex иклюдительно как показано на рис. 7.3. Длина подстроки будет равна endIndex - beginIndex. Если индексы отрицательны, индекс endIndex больше длины строки или beginIndex больше чем endIndex , то возникает исключительная ситуация.

Рассмотрим следующий пример:

```
String message = "Welcome to Java".substring(0, 11) + "HTML";
```

В результате message = "Welcome to HTML".

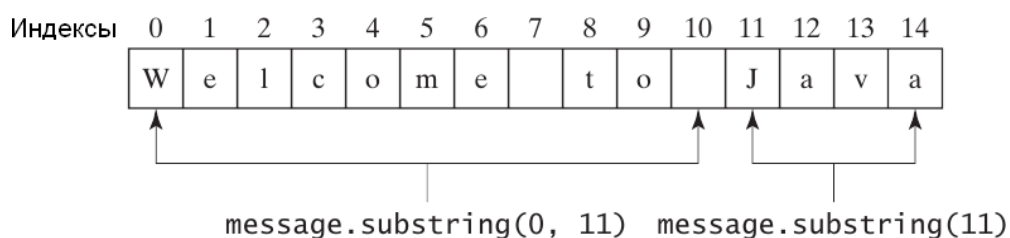


Рисунок 7.2 – Метод substring() выделяет подстроку из строки

Запомните! Если beginIndex = endIndex, то метод substring(beginIndex, endIndex) вернет пустую строку с длиной 0. Если beginIndex > endIndex, то это вызовет ошибки времени выполнения (runtime error).

Пример 7.6 *Применение методов для извлечения подстроки из строки.*

```
public class SubString
{
    public static void main( String[] args )
    {
        String letters = "abcdefghijklmabcdefghijklm";
        System.out.printf( "Подстрока от индекса 20 и до конца
                           строки \"%s\"\\n", letters.substring(20));
        System.out.printf( \"%s \"%s\"\\n", "Подстрока от индекса 3
                           включительно до индекса 6 исключительно",
                           letters.substring(3,6));
    }
}
```

Результат:

Подстрока от индекса 20 и до конца строки "hijklm"
Подстрока от индекса 3 включительно до индекса 6 исключительно "def"

7.1.7 Модификация, замена и разбиение строк

Методы класса `String` обеспечивающие модификацию, замену и разбиение строк приведены в табл.7.4.

Таблица 7.4 – Методы класса `String` для модификации, замены и разбиения строк

Метод	Описание
<code>String toLowerCase ()</code>	Возвращает новую строку, в которой все символы переведены в нижний регистр, т. е. сделаны строчными.
<code>String toUpperCase ()</code>	Возвращает новую строку, в которой все символы переведены в верхний регистр, т. е. сделаны прописными.
<code>String trim()</code>	Возвращает новую строку, в которой удалены все начальные и конечные символы с кодами, не превышающими <code>"\u0020"</code> (пробелы, управляющие символы...).
<code>String replace (char oldChar, char newChar)</code>	Возвращает новую строку, в которой все вхождения символа <code>oldChar</code> заменены символом <code>newChar</code> . Если символа <code>oldChar</code> в строке нет, то возвращается ссылка на исходную строку.
<code>String replaceFirst (String oldString,</code>	Возвращает новую строку, в которой

String newString)	первое вхождение подстроки oldString заменено подстрокой newString.
String replaceAll (String oldString, String newString)	Возвращает новую строку, в которой все вхождения подстроки oldString заменены подстрокой newString. Если подстроки oldString в строке нет, то возвращается ссылка на исходную строку
String[] split (String delimiter)	Возвращает массив строк, состоящий из лексем, полученных из исходной строки с учетом разделителя delimiter.

После того, как строка `String` создана, его содержание не может быть изменено. Методы `toLowerCase()`, `toUpperCase()`, `trim()`, `replace()`, `replaceFirst()` и `replaceAll()` возвращают новую строку, полученную из исходной строки (без изменения исходной строки!). Методы `toLowerCase()` и `toUpperCase()` возвращают новую строку путем преобразования всех символов в строке в строчные или прописные. Метод `trim()` возвращает новую строку путем устранения начальных и конечных символов пробела в исходной строке. Несколько вариантов методов `replace()` обеспечивают различные способы замены символа или подстроки в строке на новый символ или новую подстроку.

Результаты выполнения некоторых методов приведен в табл. 7.5.

Таблица 7.5 – Результат выполнения методов преобразования и замены строк

Метод	Возвращаемое значение
"Welcome". <code>toLowerCase()</code>	welcome
"Welcome". <code>toUpperCase()</code>	WELCOME
"\t Good Night \n". <code>trim()</code>	Good Night
"Welcome". <code>replace('e', 'A')</code>	WAlcomA
"Welcome". <code>replaceFirst("e", "AB")</code>	WABlcome
"Welcome". <code>replace("e", "AB")</code>	WABlcomAB
"Welcome". <code>replace("el", "AB")</code>	WABcome

Метод `split()` может быть использован для извлечения лексем из строки с учетом указанных разделителей. Как, например, в следующем коде:

```
String[] tokens = "Java#HTML#Perl".split("#");
for (int i = 0; i < tokens.length; i++)
    System.out.print(tokens[i] + " ");
```

На экране получим результат:

7.1.8 Преобразование между строками и массивами

Строки – это не массивы, но строка может быть преобразована в массив, и наоборот. Чтобы преобразовать строку в массив символов, используют метод `toCharArray()`. Например, в следующем объявлении преобразуется строка "Java" в массив:

```
char[] chars = "Java".toCharArray();
```

Таким образом, символ `chars[0]` равен `J`, `chars[1]` равен `a`, `chars[2]` равен `v`, а `chars[3]` – это `a`.

Можно также использовать метод `getChars (int srcBegin, int srcEnd, char[] dst, int dstBegin)`, чтобы скопировать из строки подстроку с индекса `srcBegin` до индекса `srcEnd-1` в массив символов `dst`, начиная с индекса `dstBegin`. Например, следующий код копирует подстроку "3720" в "CS3720" с индекса 2 до индекса 6-1 в символьный массив `dst`, начиная с индекса 4.

```
char[] dst = {'J', 'A', 'V', 'A', '1', '3', '0', '1'};
"CS3720".getChars(2, 6, dst, 4);
```

В результате массив `dst` станет равным `{'J', 'A', 'V', 'A', '3', '7', '2', '0'}`

Для преобразования массива символов в строку, можно также использовать конструктор `String(char [])` или метод `valueOf(char[])`. Например, как в следующих примерах:

```
String str= new String(new char[]{"J", 'a', 'v', 'a'});
String str= String.valueOf(new char[]{"J", 'a', 'v', 'a'});
```

7.1.9 Преобразование символов и чисел в строки

Статический метод `valueOf()` может быть использован для преобразования массива символов в строку. Имеется несколько перегруженных вариантов метода `valueOf()` с параметрами различных типов: `char`, `double`, `long`, `int` и `float`, которые могут быть использованы для преобразования символов и числовых значений в строки, как это показано в табл.7.6.

Таблица 7.6 – Перегруженные варианты метода `valueOf()`

Метод	Описание
<code>String valueOf (char c)</code>	Возвращает строку, состоящую из символа <code>c</code> .
<code>String valueOf (char[] data)</code>	Возвращает строку, состоящую из символов массива <code>data</code> .
<code>String valueOf (double d)</code>	Преобразует значение типа <code>double</code> в строку

	String.
String.valueOf (float f)	Преобразует значение типа float в строку String.
String.valueOf (int i)	Преобразует значение типа int в строку String.
String.valueOf (long l)	Преобразует значение типа long в строку String.
String.valueOf (boolean b)	Преобразует значение типа boolean в строку String.

Обратите внимание! Вы можете использовать методы `Double.parseDouble(str)` или `Integer.parseInt(str)`, чтобы преобразовать строку в значения типа `double` или `int`. `Double` и `Integer` – это два класса, расположенные в пакете `java.lang`.

Пример 7.7 *Применение методов для преобразования символов и чисел в строки.*

```
public class StringValueOf
{
    public static void main( String[] args )
    {
        char[] charArray = { 'a', 'b', 'c', 'd', 'e', 'f' };
        boolean booleanValue = true;
        char characterValue = 'Z';
        int integerValue = 7;
        long longValue = 10_000_000_000L;
        float floatValue = 2.5f;
        double doubleValue = 33.333;
        Object objectRef = "hello";
        System.out.printf("char array=%s\n",
                          String.valueOf(charArray));
        System.out.printf( "part of char array = %s\n",
                          String.valueOf(charArray, 3, 3) );
        System.out.printf("boolean=%s\n",
                          String.valueOf(booleanValue));
        System.out.printf("char=%s\n",
                          String.valueOf(characterValue));
        System.out.printf("int=%s\n",
                          String.valueOf(integerValue));
        System.out.printf("long=%s\n",
                          String.valueOf(longValue));
        System.out.printf("float=%s\n",
                          String.valueOf(floatValue));
        System.out.printf("double=%s\n",
```

```
        String.valueOf(doubleValue));  
System.out.printf("Object=%s",  
        String.valueOf(objectRef));  
    }  
}
```

Результат:

```
char array=abcdef  
part of char array = def  
boolean=true  
char=Z  
int=7  
long=10000000000  
float=2.5  
double=33.333  
Object=hello
```

7.2 Классы StringBuilder и StringBuffer

Классы `StringBuilder` и `StringBuffer` аналогичны классу `String` за исключением того, что класс `String` представляет неизменяемые последовательности символов постоянной длины, а классы `StringBuilder` и `StringBuffer` представляют расширяемые и доступные для изменений последовательности символов. Классы `StringBuilder` и `StringBuffer` являются более гибкими, чем `String`. Они позволяют добавлять, вставлять символы и подстроки в середину либо добавлять их в конец строки, в то время как длина объекта `String` фиксируется при создании.

Класс `StringBuilder` практически идентичен классу `StringBuffer`, однако между ними есть одно важное отличие: класс `StringBuilder` не синхронизирован, а значит, не является безопасным в отношении потоков. В случаях, когда к изменяемой строке обращаются несколько потоков, и нет никакой внешней синхронизации, следует использовать класс `StringBuffer`, а не класс `StringBuilder`. Однако использование класса `StringBuilder` является более эффективным, в условиях однопоточной программы. Конструкторы и методы в `StringBuffer` и `StringBuilder` практически одинаковые. В этом разделе рассмотрим `StringBuilder`. Вы можете заменить `StringBuilder` во всех примерах данного раздела на `StringBuffer` и результат будет тот же.

7.2.1 Конструкторы класса StringBuilder

Класс `StringBuilder` имеет три конструктора и более 30 методов для создания и изменения строк. Конструкторы класса `StringBuilder` приведены в табл. 7.7.

Создать объект класса `StringBuilder` можно только конструкторами. Первоначально созданный объект имеет буфер определенной емкости

(capacity), по умолчанию достаточной для хранения 16 символов. Емкость можно задать в конструкторе объекта.

Как только буфер начинает переполняться, его емкость автоматически увеличивается, чтобы вместить новые символы.

В любое время емкость буфера можно увеличить, обратившись к методу `ensureCapacity(int minCapacity)`.

Этот метод изменит емкость, только если `minCapacity` будет больше длины хранящейся в объекте строки. Емкость будет увеличена по следующему правилу. Пусть емкость буфера равна N . Тогда новая емкость будет равна $\text{Max}(2 * N + 2, \text{minCapacity})$.

Таким образом, емкость буфера нельзя увеличить менее чем вдвое.

Количество символов в строке можно узнать, как и для объекта класса `String`, методом `length()`, а емкость — методом `capacity()`.

Таблица 7.7 – Конструкторы класса `java.lang.StringBuilder`

Конструктор	Описание
<code>StringBuilder()</code>	Создает пустую строку емкостью, равной 16 символов
<code>StringBuilder(int capacity)</code>	Создает строку с указанной емкостью <code>capacity</code> .
<code>StringBuilder(String s)</code>	Создает строку емкостью <code>s.length() + 16</code> , содержащую строку <code>s</code> .

Методом `setLength(int newLength)` можно установить любую длину строки.

Если она окажется больше текущей длины, то дополнительные символы будут равны `'\u0000'`. Если она будет меньше текущей длины, то строка будет обрезана, последние символы потеряются, точнее, будут заменены символом `'\u0000'`. Емкость при этом не изменится.

Если число `newLength` окажется отрицательным, возникнет исключительная ситуация.

Дополнительные методы класса `StringBuilder` для изменения и получения свойств строки приведены в табл. 7.8.

Таблица 7.8 – Методы класса `java.lang.StringBuilder` для изменения и получения свойств строки

Метод	Описание
<code>String toString()</code>	Возвращает строку, представляющую данные в этой последовательности
<code>int capacity()</code>	Возвращает емкость строки
<code>char charAt(int index)</code>	Возвращает символ с указанным индексом
<code>int length()</code>	Возвращает количество символов в строке

<code>void setLength (int newLength)</code>	Устанавливает новую длину строки
<code>void ensureCapacity(int minCapacity)</code>	Устанавливает емкость и при этом гарантирует, что емкость, по крайней мере, равна указанному минимуму
<code>String substring(int startIndex)</code>	Возвращает подстроку, начиная с индекса <code>startIndex</code>
<code>String substring (int startIndex, int endIndex)</code>	Возвращает подстроку с индекса <code>startIndex</code> до индекса <code>endIndex-1</code>
<code>void setCharAt (int index, char ch)</code>	Символ с указанным индексом <code>index</code> , устанавливается в <code>ch</code> .
<code>void trimToSize()</code>	Уменьшает емкость до фактического размера строки

В коде примера 7.8 демонстрируется применение отдельных методов из табл. 7.8.

Пример 7.8 *Применение методов `length()`, `capacity()`, `setLength()` и `ensureCapacity()`*

Объяснение: начальная емкость строки "Hello, how are you?" равна 35, а длина равна 19. Емкость строки получена в результате сложения длины строки +16. Далее в программе емкость строки увеличена до 75. Помните, что динамическое расширение емкости `StringBuilder` может занять довольно длительное время. Выполнение большого количества этих операций может привести к снижению производительности приложения. В программе также используется метод `setLength()`, который усекает длину `StringBuilder` до 10. Если заданная длина меньше, чем текущее количество символов в `StringBuilder`, то буфер усекается до заданной длины (например, лишние символы отбрасываются). Если указанная длина больше, чем текущее число символов, то добавляются нулевые символы (символы с числовым представлением 0) до тех пор, пока общее количество символов в `StringBuilder` не станет равна указанной длине.

```
public class StringBuilderCapLen
{
    public static void main( String[] args )
    {
        StringBuilder buffer =
            new StringBuilder("Hello, how are you?");

        System.out.printf("buffer=%s\nlength=%d\ncapacity=%d\n\n",
            buffer.toString(), buffer.length(), buffer.capacity());
    }
}
```

```

buffer.ensureCapacity(75);
System.out.printf("New capacity=%d\n\n",
                  buffer.capacity());

buffer.setLength(10);
System.out.printf( "New length = %d\nbuffer = %s\n",
buffer.length(), buffer.toString() );
}
}

```

Результат:

```

buffer = Hello, how are you?
length = 19
capacity = 35

```

```

New capacity = 75

```

```

New length = 10
buffer = Hello, how

```

7.2.2 Добавление, удаление и замена подстроки

В классе `StringBuilder` есть десять методов `append()`, добавляющих подстроку в конец строки (см. табл. 7.9). Они не создают новый экземпляр строки, а возвращают ссылку на ту же самую, но измененную строку.

Основной метод `append (String s)` присоединяет строку `s` в конец данной строки. Если ссылка `s == null`, то добавляется строка `"null"`.

Шесть методов `append (aPrimitiveType value)` добавляют примитивные типы `boolean`, `char`, `int`, `long`, `float`, `double`, преобразованные в строку.

Два метода присоединяют к строке массив и подмассив символов `data`, преобразованные в строку: `append (char [] data)` и `append (char [] data, int offset, int len)`.

Десятый метод добавляет просто объект `append (Object obj)`. Перед этим объект `obj` преобразуется в строку своим методом `toString()`.

На самом деле, компилятор может использовать методы добавления класса `StringBuilder` (или `StringBuffer`) выполняя `+` и `+=` операторы конкатенации строк. Так, например, в следующем коде

```

String string1 = "hello";
String string2 = "BC";
int value = 22;
String s = string1 + string2 + value;

```

конкатенируются `"hello"`, `"BC"` и `22`. Компилятор выполнит компиляцию следующим образом:

```

String s = new StringBuilder().append( "hello" ).append(
"BC" ).append( 22 ).toString();

```


Будет создана пустая строка `StringBuilder`, затем к ней будут добавлены строки "hello", "BC" и 22. Затем с помощью метода `toString()` класса `StringBuilder` строка будет преобразована в объект `String` и связана с ссылкой `String s`.

Соответственно оператор `s+="!";` может быть выполнен следующим образом:

```
s=new StringBuilder().append(s).append("!").toString();
```

Таблица 7.9 – Методы класса `java.lang.StringBuilder` для изменения содержания строки

Метод	Описание
<code>StringBuilder append(char[] data)</code>	Добавляет массив символов <code>data</code> в конец строки
<code>StringBuilder append(char[] data, int offset, int len)</code>	Добавляет подмассив <code>data</code> в конец строки
<code>StringBuilder append(aPrimitiveType value)</code>	Добавляет значение примитивного типа <code>value</code> , преобразованное в строку
<code>StringBuilder append(String s)</code>	Добавляет строку <code>s</code> в конец текущей строки.
<code>StringBuilder insert(int index, char[] data, int offset, int len)</code>	Вставляет подмассив <code>data</code> в строку перед указанным индексом <code>index</code>
<code>StringBuilder insert(int offset, char[] data)</code>	Вставляет массив символов <code>data</code> в строку перед указанным индексом <code>index</code>
<code>StringBuilder insert(int offset, aPrimitiveType value)</code>	Вставляет в строку перед указанным индексом <code>offset</code> значение примитивного типа, преобразованное в строку
<code>StringBuilder insert(int offset, String s)</code>	Вставляет строку <code>s</code> в текущую строку перед указанным индексом <code>offset</code>
<code>StringBuilder delete (int startIndex, int endIndex)</code>	Удаляет символы с индекса <code>startIndex</code> включительно до <code>endIndex</code> исключительно
<code>StringBuilder deleteCharAt(int index)</code>	Удаляет символ с указанным индексом <code>index</code> .
<code>StringBuilder replace (int startIndex, int endIndex, String s)</code>	Заменяет символы в строке с индекса <code>startIndex</code> до индекса <code>endIndex-1</code> на указанную строку <code>s</code> .
<code>StringBuilder reverse()</code>	Меняет порядок расположения символов в строке на обратный

Десять методов `insert()` предназначены для вставки строки, указанной параметром метода, в данную строку. Место вставки задается

первым параметром `offset`. Это индекс элемента строки, перед которым будет сделана вставка. Он должен быть неотрицательным и меньше длины строки, иначе возникнет исключительная ситуация. Строка раздвигается, емкость буфера при необходимости увеличивается. Методы возвращают ссылку на ту же преобразованную строку.

Основной метод `insert (int offset, String s)` вставляет строку `s` в данную строку перед ее символом с индексом `offset`. Если ссылка `s == null` вставляется строка `"null"`.

Например, после выполнения

```
String s = new StringBuffer("Это большая строка").  
insert(4, "не").toString();
```

получим: `s = "Это небольшая строка"`.

Шесть методов `insert(int offset, aPrimitiveType value)` вставляют примитивные типы `boolean`, `char`, `int`, `long`, `float`, `double`, преобразованные в строку.

Два метода вставляют массив и подмассив символов `data`, преобразованные в строку:

```
insert(int offset, char[] data)  
insert(int index, char[] data, int offset, int len)
```

Десятый метод вставляет просто объект:

```
insert(int offset, Object obj)
```

Объект `obj` перед добавлением преобразуется в строку своим методом `toString()`.

Метод `delete (int startIndex, int endIndex)` удаляет из строки символы, начиная с индекса `startIndex` включительно до индекса `endIndex` исключительно, если `endIndex` больше длины строки, то до конца строки.

Например, после выполнения

```
String s = new StringBuffer("Это небольшая строка").  
delete(4, 6).toString();
```

получим `s = "Это большая строка"`.

Если `startIndex` отрицательный, больше длины строки или больше `endIndex`, возникает исключительная ситуация. Если `startIndex == endIndex`, удаление не происходит.

Метод `deleteCharAt (int index)` удаляет символ с указанным индексом `index`. Длина строки уменьшается на единицу. Если индекс `index` отрицателен или больше длины строки, возникает исключительная ситуация.

Метод `replace(int startIndex, int endIndex, String s)` удаляет символы из строки, начиная с индекса `startIndex` включительно до индекса `endIndex` исключительно, если `endIndex` больше длины строки, то до конца строки, и вставляет вместо них строку `s`.

Если `startIndex` отрицательный, больше длины строки или больше `endIndex`, возникает исключительная ситуация. Метод `replace()` – это последовательное выполнение методов `delete()` и `insert()`.

Метод `reverse()` меняет порядок расположения символов в строке на обратный.

Например, после выполнения

```
String s = new StringBuffer("Это небольшая строка"),  
reverse().toString();  
получим s = "акортс яшьлобен отЭ".
```

7.3 Класс Character

Как вам уже известно, из предыдущих глав, в Java имеются восемь примитивных типов: `boolean`, `byte`, `short`, `int`, `long`, `char`, `float` и `double`. Эти типы данных не являются частью объектной иерархии и используются из соображений производительности. Они передаются по значению в методы и не могут быть переданы по ссылке. Рано или поздно возникнет необходимость в объектном представлении одного из примитивных типов.

Чтобы сохранить примитивный тип в одном из этих классов, необходимо поместить его в оболочку класса. Чтобы удовлетворить эту потребность, язык Java предлагает классы, которые соответствуют каждому примитивному типу: `Boolean`, `Character`, `Double`, `Float`, `Byte`, `Short`, `Integer` и `Long`. По сути, эти классы инкапсулируют примитивные типы в классы (или, что то же самое, помещают примитивные типы в оболочки классов). Все эти классы-оболочки находятся в пакете `java.lang`, и содержат полезные методы обработки примитивных значений. В данном разделе рассмотрим класс `Character`.

Класс `Character` – это простая оболочка для типа `char`. Класс имеет один конструктор и несколько методов, описанных в табл. 7.10. Объект `Character` можно создать из символа, например:

```
Character character = new Character('a');
```

Здесь символ `'a'` будет помещен в оболочку создаваемого объекта класса `Character`.

Таблица 7.10 – Конструктор и методы класса `java.lang.Character`

Метод	Описание
<code>Character(char value)</code>	Конструктор объекта <code>Character</code> с параметром – символьным значением
<code>char charValue()</code>	Возвращает символ, содержащийся в объекте класса <code>Character</code>
<code>int compareTo(Character anotherCharacter)</code>	Возвращает нуль, если вызывающий объект и <code>anotherCharacter</code> эквивалентны, и

	отрицательное значение – если вызывающий объект содержит меньшее значение. В противном случае возвращает положительное значение.
<code>boolean equals(Character anotherCharacter)</code>	Возвращает <code>true</code> , если вызывающий объект и <code>anotherCharacter</code> эквивалентны, и <code>false</code> в противном случае.
<code>boolean isDefined(char ch)</code>	Возвращает значение <code>true</code> , если символ <code>ch</code> определен в Unicode. В противном случае возвращает значение <code>false</code>
<code>boolean isDigit(char ch)</code>	Возвращает значение <code>true</code> , если символ <code>ch</code> является десятичной цифрой. В противном случае возвращает значение <code>false</code>
<code>boolean isJavaIdentifierPart(char ch)</code>	Возвращает значение <code>true</code> , если символ <code>ch</code> может быть частью идентификатора Java. В противном случае возвращает значение <code>false</code>
<code>boolean isJavaIdentifierStart(char ch)</code>	Возвращает значение <code>true</code> , если символ <code>ch</code> может быть первым символом идентификатора Java. В противном случае возвращает значение <code>false</code>
<code>boolean isLetter(char ch)</code>	Возвращает значение <code>true</code> , если символ <code>ch</code> – буква. В противном случае возвращает значение <code>false</code>
<code>boolean isLetterOrDigit(char ch)</code>	Возвращает значение <code>true</code> , если символ <code>ch</code> – буква или цифра. В противном случае возвращает значение <code>false</code>
<code>boolean isLowerCase(char ch)</code>	Возвращает значение <code>true</code> , если символ <code>ch</code> – буква в нижнем регистре. В противном случае возвращает значение <code>false</code>
<code>boolean isUpperCase(char ch)</code>	Возвращает значение <code>true</code> , если символ <code>ch</code> – символ верхнего регистра. В противном случае возвращает значение <code>false</code>
<code>char toLowerCase(char ch)</code>	Возвращает эквивалент символа <code>ch</code> в нижнем регистре
<code>char toUpperCase(char ch)</code>	Возвращает эквивалент символа <code>ch</code> в верхнем регистре

Задача 7.2. Напишите программу, которая предлагает пользователю ввести строку и подсчитывает количество вхождений каждой буквы независимо от регистра.

Объяснение: запишем алгоритм решения задачи:

1. Конвертировать все прописные буквы в строке в строчные помощью метода `toLowerCase()` класса `String`.

2. Создадим массив `counts[]` состоящий из 33 целых значений, каждое из которых будет соответствовать количеству вхождений букв алфавита. То есть, `counts[0]` – количество вхождений буквы 'а', `counts[1]` – количество вхождений буквы 'б' и так далее.

3. Для каждого символа в строке, проверяем, является ли он строчной буквой. Если да, то увеличиваем на единицу соответствующее количество в массиве `counts[]`.

В главном методе вводится строка и подсчитывается количество вхождений каждой буквы путем вызова метода `countLetters()`. Поскольку регистр букв по условию задачи игнорируется, в программе используется метод `toLowerCase()` для преобразования строки в нижний регистр. Далее уже эта новая строка передается в метод `countLetters()`, который возвращает массив из 33 элементов. Каждый элемент – это количество вхождений буквы в строку `s`. Метод проверяет каждый символ строке. Если символ является буквой, то соответствующий счетчик увеличивается на 1. Например, если символ `s.charAt(i)` является символом 'а', то соответствующий ему элемент массива `counts['а' - 'а']` (т.е. `counts[0]`). Для символа 'б' – `counts['б' - 'а']` (т.е. `counts[1]`), так как в кодировке Unicode код символа 'б' на 1 больше, чем 'а'. Для символа 'я', соответствующий ему элемент массива `counts['я' - 'а']` (т.е. `counts[32]`), так как в Unicode 'я' на 32 больше, чем 'а'.

```
import java.util.Scanner;
public class CountEachLetter {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in, "UTF-8");
        System.out.print("Введите строку: ");
        String s = input.nextLine();
        int[] counts = countLetters(s.toLowerCase());
        for (int i = 0; i < counts.length; i++) {
            if (counts[i] != 0)
                System.out.println((char)('а' + i) + " встречается " +
counts[i] + " раз");
        }
    }

    public static int[] countLetters(String s) {
        int[] counts = new int[33];
        for (int i = 0; i < s.length(); i++) {
            if (Character.isLetter(s.charAt(i)))
                counts[s.charAt(i) - 'а']++;
        }
    }
}
```

```
return counts;
    }
}
```

Результат:

Введите строку: результат выполнения программы

а	встречается	2 раз
в	встречается	1 раз
г	встречается	1 раз
е	встречается	2 раз
з	встречается	1 раз
и	встречается	1 раз
л	встречается	2 раз
м	встречается	2 раз
н	встречается	2 раз
о	встречается	2 раз
п	встречается	2 раз
р	встречается	3 раз
т	встречается	2 раз
у	встречается	1 раз
ы	встречается	2 раз
ь	встречается	1 раз
я	встречается	1 раз

7.4 Аргументы командной строки

Главный метод `main()` может принимать строковые аргументы из командной строки. Конечно, вы уже заметили в заголовке главного метода параметр `args` типа `String[]`, являющийся массивом строк. Метод `main()` похож на обычный метод с параметром, при вызове которого вы можете передать аргументы из командной строки и использовать их как обычный массив в теле метода. Например, так как в следующем коде:

```
public class TestMain {
    public static void main (String[] args)
    {
        for (int i = 0; i < args.length; i++)
            System.out.println(args[i]);
    }
}
```

Передать строки в главный метод можно из командной строки при запуске программы. Например, следующая командная строка запускает программу `TestMain` с тремя строками: `arg0`, `arg1` и `arg2`:

```
Java TestMain arg0 arg1 arg2
```

`arg0`, `arg1` и `arg2` являются строками, но они могут не заключаться в двойные кавычки, однако должны быть разделены пробелом. Если строка содержит пробелы, то ее следует обязательно заключить в двойные кавычки. Рассмотрим следующую команду:

```
Java TestMain "First num" alpha 53
```

Программа запускается с тремя аргументами-строками: `First num`, `alpha` и `53`. Строка `First num` заключена в двойные кавычки. Обратите внимание, что на самом деле `53` обрабатывается как строка. Вы можете записывать `"53"` вместо `53` в командной строке.

Если вы работаете в IDE BlueJ, то аргументы командной строки передаются при вызове главного метода в окне, вид которого изображен на рис.7.3. При этом аргументы должны быть заключены в двойные кавычки и разделены запятыми.

Когда главный метод вызывается, интерпретатор Java создает массив для хранения аргументов командной строки и передает ссылку на массив `args`.



Рисунок 7.3 – Передача аргументов командной строки в IDE BlueJ

Если вы запускаете программу без каких-либо аргументов, создается пустой массив строк `args[]` с нулевой длиной.

7.5 Синтаксический разбор строки

Задача разбора введенного текста – *парсинг* (parsing) – вечная задача программирования, наряду с сортировкой и поиском. Написана масса программ-парсеров (parser), разбирающих текст по различным признакам. Есть даже программы, генерирующие парсеры по заданным правилам разбора: YACC, LEX и др. Но задача остается актуальной. Рассмотрим, какие средства предоставляет программистам Java для разбора строк на отдельные лексемы.

7.5.1 Метод `split()` класса `String`

В разделе 7.1.6 текущей главы был рассмотрен метод `split(String delimiter)` класса `String`, который возвращает массив строк, состоящий из лексем, полученных из исходной строки с учетом разделителя

delimiter. Рассмотрим пример использования данного метода для разбора строки.

Пример 7.9. *Применение метода `split()` для разбора строки на лексемы.*

```
import java.util.Scanner;
public class TokenTest
{
    public static void main( String[] args )
    {
        Scanner scanner = new Scanner(System.in, "UTF-8");
        System.out.println("Введите строку:");
        String line = scanner.nextLine();
        String[] tokens = line.split(" ");
        System.out.printf( "Количество слов: %d\nСлова:\n",
tokens.length);
        for(String token : tokens){
            System.out.println(token);
        }
    }
}
```

Результат:

```
Введите строку:
Это очень большая строка для разбиения на лексемы
Количество слов: 8
Слова:
Это
очень
большая
строка
для
разбиения
на
лексемы
```

Рассмотрим задачу, в которой будет использоваться рассмотренный метод `split()` для парсинга введенного арифметического выражения.

Задача 7.3. *Напишите программу - калькулятор, выполняющую операции над целыми числами. Программа получает выражение в одной строке аргумента. Выражение состоит из целого числа, оператора и второго целого числа. Например, чтобы сложить два целых числа, нужно передать строку: "2 + 3", при этом программа должна выдать следующий результат: 2 + 3 = 5.*

Объяснение: по условию задачи выражение для вычисления необходимо передавать в главный метод в виде одного аргумента командной строки. В таком случае Массив `args[]` будет содержать только один элемент – `args[0]` и `args.length = 1`.

Запишем алгоритм решения задачи:

1. Используя `args.length` определим, является ли переданное выражение одним аргументом командной строки. Если нет, то завершим выполнение программы, используя оператор `System.exit(1)`.

2. Разделим выражение в строке `args[0]` на три лексемы (слова) `tokens[0]`, `tokens[1]` и `tokens[2]`. В качестве разделителя будем использовать метод `Integer.parseInt()`, который преобразует строку в число. При этом строка должна состоять из цифр. Если это не так, то программа будет завершена с ошибкой.

3. Выполним арифметические операции над операндами `tokens[0]` и `tokens[2]` с использованием оператора `tokens[1]`.

```
public class Calculator {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Используйте для ввода:
                                \"operand1 operator operand2\");
            System.exit(1);
        }
        int result = 0;
        String[] tokens = args[0].split(" ");
        switch(tokens[1].charAt(0)){
            case '+': result = Integer.parseInt(tokens[0]) +
                            Integer.parseInt(tokens[2]);
                        break;
            case '-': result = Integer.parseInt(tokens[0]) -
                            Integer.parseInt(tokens[2]);
                        break;
            case '*': result = Integer.parseInt(tokens[0]) *
                            Integer.parseInt(tokens[2]);
                        break;
            case '/': result = Integer.parseInt(tokens[0]) /
                            Integer.parseInt(tokens[2]);
        }

        System.out.println(tokens[0] + ' ' + tokens[1] + ' ' +
            tokens[2] + " = " + result);
    }
}
```

В коде задачи 7.3 был использован метод `Integer.parseInt()`, который находится в класса `java.lang.Integer` и предназначен для преобразования строки в число. Другие классы-оболочки для примитивных типов: `Short`, `Long`, `Float` и `Double` также содержат статические методы:

```
parseShort(String s)
parseLong(String s)
```

```
parseFloat(String s)
parseDouble(String s),
```

которые возвращают числовые эквиваленты строки `s`.

7.5.2 Класс `StringTokenizer`

В пакет `java.util` входит простой класс `StringTokenizer`, облегчающий разбор строк. Класс `StringTokenizer` небольшой, в нем всего три конструктора и шесть методов, наиболее важные из которых, приведены в табл. 7.11.

Таблица 7.11 – Конструктор и методы класса `java.util. StringTokenizer`

Конструктор/Метод	Описание
<code>StringTokenizer(String s)</code>	Конструктор создает объект, готовый разбить строку <code>s</code> на лексемы, разделенные пробелами, символами табуляций <code>'\t'</code> , перевода строки <code>'\n'</code> и возврата каретки <code>'\r'</code> . Разделители не включаются в число слов.
<code>StringTokenizer (String s, String delimiters)</code>	Конструктор задает разделители вторым параметром <code>delimiters</code> , например: <code>StringTokenizer("Казнить,нельзя:пробелов-нет", "\t\n\r,:-");</code> здесь первый разделитель – пробел. Потом идут символ табуляции, символ перевода строки, символ возврата каретки, запятая, двоеточие, дефис. Порядок расположения разделителей в строке <code>delimiters</code> не имеет значения. Разделители не включаются в число слов.
<code>StringTokenizer(String s, String delimiters, boolean flag)</code>	Конструктор позволяет включить разделители в число слов. Если параметр <code>flag</code> равен <code>true</code> , то разделители включаются в число слов, если <code>false</code> – нет. Например: <code>StringTokenizer("a - (b + c) / b * c", "\t\n\r+*-/()", true);</code>
<code>String nextToken ()</code>	Возвращает в виде строки следующую лексему
<code>boolean hasMoreTokens ()</code>	Возвращает <code>true</code> , если в строке еще есть слова, и <code>false</code> , если слов больше нет
<code>int countTokens ()</code>	Возвращает количество оставшихся в строке лексем
<code>String nextToken(String newDelimiters)</code>	Позволяет "на ходу" менять разделители. Следующая лексема будет выделена по

	новым разделителям <code>newDelimiters</code> ; новые разделители действуют далее вместо старых разделителей, определенных в конструкторе или предыдущем методе <code>nextToken()</code> .
--	--

В разборе строки на лексемы (слова) активно участвуют два метода: метод `nextToken()`, который возвращает в виде строки следующее слово и логический метод `hasMoreTokens()`, который возвращает `true`, если в строке еще есть слова, и `false`, если слов больше нет. Метод `countTokens()` возвращает число оставшихся слов.

Схема парсинга очень проста и приведена ниже:

```
String s = "Строка, которую мы хотим разобрать на слова";
StringTokenizer st = new StringTokenizer(s, " \t\n\r,.");
while(st.hasMoreTokens()){
    // Получаем слово и что-нибудь делаем с ним, например,
    // просто выводим на экран
    System.out.println(st.nextToken());
}
```

Важное дополнение! Описание стандартных средств Java для работы со строками был бы не полный без упоминания о регулярных выражениях и классах `Pattern` и `Matcher`. *Регулярное выражение* (Regular Expressions) представляет собой строку, которая описывает шаблон поиска соответствующих символов в другой строке. Такие выражения используются для проверки верности введенных данных и обеспечивают ввод данных в определенном заданном формате. Например, почтовый индекс должен состоять из пяти цифр, и фамилия должна содержать только буквы и пробелы. Одно из применений регулярных выражений – это разработка компилятора. Часто большие и сложные регулярные выражения используются для проверки синтаксиса программы. Если код программы не совпадает с регулярным выражением, компилятор знает, что есть синтаксические ошибки в коде. За неимением времени, в данном курсе мы не будем изучать регулярные выражения. Те, кому интересна данная тема могут найти информацию о регулярных выражениях в книгах по Java, а также изучая многочисленные ресурсы на информационных просторах Интернета.

7.6 Текстовый ввод-вывод

Базовые классы для ввода/вывода расположены в пакете `java.io`. Поэтому для работы с ними нужно записать `import java.io.*`; а в программе объявить экземпляры соответствующих классов.

Консольный ввод с помощью методов класса `Scanner` был рассмотрен нами в главе 3, здесь кратко поговорим о консольном вводе с помощью

стандартных средств пакета `java.io`, которые были в Java, начиная с ее самой первой версии.

Для ввода с клавиатуры в программе следует объявить экземпляры следующих классов:

```
InputStreamReader is = new InputStreamReader(System.in);
BufferedReader st = new BufferedReader(is);
```

Для ввода данных типа `String` следует использовать метод `readLine()` класса `BufferedReader`.

Ниже приведен пример ввода строки с консоли:

```
import java.io.*;
class Input {
    public static void main(String[] args){
        InputStreamReader is = new InputStreamReader(System.in);
        BufferedReader bis = new BufferedReader(is);
        try {
            System.out.println("Введите строку:");
            String s = bis.readLine();
            System.out.println("Введенная строка "+s);
        } catch (IOException e)
        {System.out.println("ошибка ввода "+ e);}
    }
}
```

Обратите внимание на то, что ввод данных следует обязательно осуществлять в обработчике исключительных ситуаций `try-cath` (так как это сделано в примере). Компилятор следит за этим и в противном случае выдаст ошибку компиляции.

При вводе символов и числовых данных придется проводить преобразование из типа `String` в соответствующий тип. Для этого могут быть использованы следующие методы классов-оболочек примитивных типов: метод `valueOf(String s)`, который возвращает объект класса-оболочки, соответствующий цифровой строке и метод `xxxValue()`, который преобразует значение этого класса к базовому примитивному типу `xxx`. Например:

```
Integer.valueOf(s).intValue();
Long.valueOf(s).longValue();
Float.valueOf(s).floatValue();
Double.valueOf(s).doubleValue();
```

Для типа `char` используют метод, позволяющий преобразовать строку в массив символов `char[]`:

```
s.toCharArray();
```

Ввести данные соответствующих примитивных числовых типов можно с помощью подобного кода:

```
System.out.println("введите число a:");
String s = st.readLine();
Double number = Double.valueOf(s).doubleValue();
```

Для преобразования строки в число можно также использовать рассмотренные ранее статические методы классов-оболочек Integer, Short, Long, Float и Double:

```
parseInt(String s)
parseShort(String s)
parseLong(String s)
parseFloat(String s)
parseDouble(String s)
```

которые возвращают числовые эквиваленты строки s.

Пример 7.10. Ввод данных с клавиатуры с помощью стандартных средств Java

```
import java.io.*;
class Vvod{
public static void main (String[] args){
double a,b;
int c;
InputStreamReader is = new InputStreamReader(System.in);
BufferedReader bis = new BufferedReader(is);
try {
System.out.println("Введите число a:");
String numberStr = bis.readLine();
a = Double.valueOf(numberStr).doubleValue();

System.out.println("Введите число b:");
numberStr = bis.readLine();
b = Double.valueOf(numberStr).doubleValue();

System.out.println("Введите число a:");
int a = Integer.parseInt(br.readLine());

} catch (IOException e)
{System.out.println("ошибка ввода "+ e);}

System.out.println((a+b)*c);
}}
```

7.6.1 Класс File

Данные, хранящиеся в программе, являются временными, они теряются, когда программа завершается. Для постоянного хранения данные необходимо сохранить в файл на диске или на других постоянных устройствах хранения данных. Файл можно открыть и прочитать уже после завершения программы. Поскольку данные хранятся в файлах, в этом разделе рассмотрим, как можно использовать класс File для получения свойств файла (каталога), удаления и переименования файла

(каталога), а также создания каталога. В следующем разделе рассмотрим, как осуществляется чтение/запись из/в текстовые файлы.

Файлы размещаются в каталоге в файловой системе. Абсолютное имя файла (или полное имя) содержит имя файла с указанием полного пути. Например, `c:\book\Welcome.java` – это абсолютное имя для файла `Welcome.java` в операционной системе Windows. Здесь `c:\book` путь к каталогу, в котором хранится файл. Абсолютное имя файла зависит от платформы. Так в UNIX, абсолютное имя файла может иметь вид `/home/liang/book/Welcome.java`, где `/home/liang/book` – это путь к каталогу для файла `Welcome.java`.

Относительное имя файла – это его имя по отношению к текущему рабочему каталогу. Например, `Welcome.java` является относительным именем файла, если текущий рабочий каталог `c:\book`, при этом абсолютное имя файла будет `c:\book\Welcome.java`.

Класс `File` содержит методы для получения свойств файлов и директорий и для переименования и удаления файлов и каталогов, которые описаны в табл. 7.12. Тем не менее, *класса `File` не содержит методы для чтения и записи содержимого файла.*

Имя файла – это строка. Класс `File` является классом-оболочкой для имени файла и пути к каталогу. Например, `new File("c:\\book")` создает объект `File` для каталога `c:\book`, а `new File("c:\\book\\test.dat")` создает объект `File` для файла `c:\book\test.dat`, созданного в Windows. Можно использовать метод `isDirectory()` класса `File` для того, чтобы проверить, является ли объект каталогом, а метод `isFile()` для проверки является ли объект файлом.

Таблица 7.12 – Конструктор и методы класса `java.io.File`

Конструктор/Метод	Описание
<code>File (String pathname)</code>	Создает объект <code>File</code> для заданного пути. Путь может быть к файлу или к каталогу.
<code>File(String parent, String child)</code>	Создает объект <code>File</code> для <code>child</code> в директории <code>parent</code> . <code>child</code> может быть именем файла или подкаталога.
<code>File(File parent, String child)</code>	Создает объект <code>File</code> для <code>child</code> в директории <code>parent</code> , где <code>parent</code> является объектом <code>File</code> . В предыдущем конструкторе <code>parent</code> является строкой.
<code>boolean exists()</code>	Возвращает <code>true</code> , если файл или каталог, представленный объектом <code>File</code> , существует.
<code>boolean canRead()</code>	Возвращает <code>true</code> , если файл, представленный объектом <code>File</code> , существует и может быть прочитан.

<code>boolean canWrite()</code>	Возвращает true, если файл, представленный объектом File, существует и может быть записан.
<code>boolean isDirectory()</code>	Возвращает true, если объект File является каталогом.
<code>boolean isFile()</code>	Возвращает true, если объект File является файлом.
<code>boolean isAbsolute()</code>	Возвращает true, если объект File создан с помощью абсолютного пути.
<code>boolean isHidden()</code>	Возвращает true, если файл представленный объектом File скрытый. Точное определение «скрытый» зависит от системы. В Windows, можно пометить файл, как скрытый, в диалоговом окне свойств файла. В системах Unix, файл скрытый, если ее имя начинается с символа точка (.).
<code>String getAbsolutePath()</code>	Возвращает полное абсолютное имя файла или каталога, представленного объектом File.
<code>String getCanonicalPath()</code>	Возвращает то же, что и <code>getAbsolutePath()</code> кроме того, что удаляет избыточные имена, такие как <code>""</code> и <code>".."</code> , в имени пути, разрешает символические ссылки (в Unix), и преобразует буквы дисков в стандартном верхнем регистре (в Windows).
<code>String getName()</code>	Возвращает последнее имя в полном пути каталога или файла. Например, <code>new File("c:\\book\\test.dat").getParent()</code> вернет <code>test.dat</code> .
<code>String getPath()</code>	Возвращает полный путь каталога и имени файла. Например, <code>new File("c:\\book\\test.dat").getPath()</code> вернет <code>c:\\book\\test.dat</code> .
<code>String getParent()</code>	Возвращает полный родительский каталог текущего каталога или файла. Например, <code>new File("c:\\book\\test.dat").getParent()</code> вернет <code>c:\\book</code> .
<code>long lastModified()</code>	Возвращает время последнего изменения файла.
<code>long length()</code>	Возвращает размер файла, или 0, если он не существует или если он является каталогом.
<code>File[] listFile()</code>	Возвращает файлы из директории.
<code>boolean delete()</code>	Удаление файла или каталога. Метод

	возвращает true, если удаление прошло успешно. Если каталог содержит файлы и другие каталоги, то он не удаляется! Сначала нужно отдельно удалить все содержимое, а потом уже удалить пустой каталог.
<code>boolean renameTo(File dest)</code>	Переименование файла или каталога с указанным именем dest. Метод возвращает true, если операция выполнена успешно.
<code>boolean mkdir()</code>	Создает каталог в объекте File. Возвращает true, если каталог успешно создан.

Будьте внимательны! Разделитель в записи пути к файлу в Windows - обратная косая черта (\). Обратная косая черта является специальным символом в Java и должна быть записана в тексте в виде \\.

Конструктор, создающий экземпляр File не создает файл на машине!!!
Вы можете создать экземпляр File для любого файла независимо от того, существует он или нет.

В примере 7.11 показано, как создать объект и использовать методы класса File для получения его свойств. Программа создает объект File для файла `MyFile.txt`. Этот файл хранится в текущем каталоге, т.е. там же где и сама программа.

Пример 7.11. Применение методов класса File.

```
public class TestFileClass {
    public static void main(String[] args) {
        java.io.File file = new java.io.File("MyFile.txt");
        System.out.println("Does it exist? "+file.exists());
        System.out.println("The file has "+file.length()+" bytes");
        System.out.println("Can it be read? "+file.canRead());
        System.out.println("Can it be written? "+file.canWrite());
        System.out.println("Is it a directory? "+file.isDirectory());
        System.out.println("Is it a file? "+file.isFile());
        System.out.println("Is it absolute? "+file.isAbsolute());
        System.out.println("Is it hidden? "+file.isHidden());
        System.out.println("Absolute path is "+
                           file.getAbsolutePath());
        System.out.println("Last modified on "+ new
                           java.util.Date(file.lastModified()));
    }
}
```

Результат:

```
Does it exist? true
The file has 10 bytes
Can it be read? true
```



```
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
Absolute path is E:\MyProgram\sss\MyFile.txt
Last modified on Sun Apr 14 20:36:17 EEST 2013
```

В разделе 6 нами было изучено понятие рекурсии. Как вы могли видеть, рекурсивные методы эффективны для решения проблем с рекурсивными структурами. Попробуем в данном разделе с помощью рекурсии решить задачу, которую трудно решить без использования рекурсии. Задача состоит в том, чтобы найти размер директории. Размер директории является суммой размеров всех файлов в директории. Директория d может содержать поддиректории. Предположим, директория содержит файлы $f_1, f_2, \dots, f_m$ и поддиректории $d_1, d_2, \dots, d_n$, как показано на рис. 7.4.

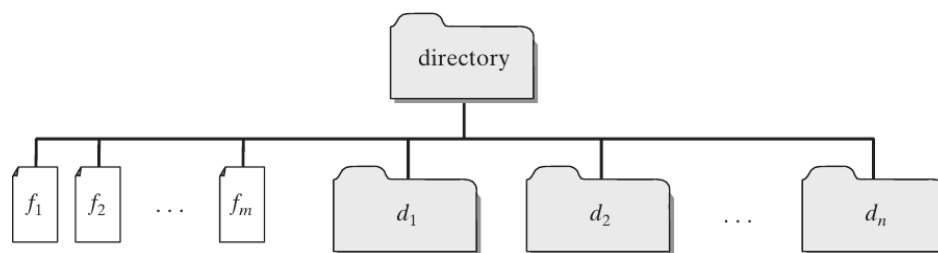


Рисунок 7.4 – Директория состоит из файлов и поддиректорий

Размер каталога можно рекурсивно определить следующим образом:

$$\text{size}(d) = \text{size}(f_1) + \text{size}(f_2) + K + \text{size}(f_m) + \text{size}(d_1) + \text{size}(d_2) + K + \text{size}(d_n)$$

Для решения задачи целесообразно использовать два метода класса File:

- `length()` возвращает размер файла.
- `listFiles()` возвращает массив объектов File из директории.

Задача 7.4. Напишите рекурсивную программу, определяющую размер директории.

Объяснение: если объект `file` представляет собой директорию, то каждый элемент (файл или поддиректории) в директории рекурсивно вызывается для получения его размера. Если объект `file` представляет собой файл, то добавляется размер файла.

При неверном вводе либо вводе несуществующей директории программа обнаружит, что она не является каталогом и вызовет метод

`file.length()`, который вернет 0. Таким образом, в этом случае метод `getSize()` вернет 0.

```
import java.io.File;
import java.util.Scanner;
public class DirectorySize {
    public static void main(String[] args) {
        System.out.print("Enter a directory or a file:");
        Scanner input = new Scanner(System.in);
        String directory = input.nextLine();
        System.out.println(getSize(new File(directory))+" bytes");
    }
    public static long getSize(File file) {
        long size = 0;
        if(file.isDirectory()) {
            File[] files = file.listFiles();
            for (int i = 0; files != null && i < files.length; i++)
            {
                size += getSize(files[i]);
            }
        }
        else {
            size += file.length();
        }
    }
    return size;
}
}
```

Результат:

```
Enter a directory or a file: c://MyProgram
18368625 bytes
```

7.6.2 Файловый ввод - вывод

В данном разделе рассмотрим использование класса `Scanner` для чтения данных из текстового файла и класса `PrintWriter` для записи данных в текстовый файл.

Объект `File` инкапсулирует свойства файла, но он не содержит методов для создания файла или для чтения/записи данных из/в файл (называемый вводом-выводом данных). Для выполнения ввода-вывода нужно создать объект, используя соответствующие классы ввода/вывода Java. Объекты содержат методы для чтения/записи данных из/в файл. Есть два типа файлов: текстовые и бинарные. В этом разделе рассмотрим работу только с текстовыми файлами.

Запись в текстовый файл. Класс `PrintWriter`. Класс `java.io.PrintWriter` может быть использован для создания файла и записи данных в текстовый файл. Для этого нужно создать объект `PrintWriter` для текстового файла следующим образом:

```
PrintWriter output = new PrintWriter(filename);
```

Затем, для объекта `PrintWriter` могут быть вызваны методы `print()`, `println()`, и `printf()` для записи данных в файл. В табл. 7.13 приведены часто используемые методы класса `PrintWriter`.

Таблица 7.13 – Конструкторы и методы класса `java.io. PrintWriter`

Конструктор / Метод	Описание
<code>PrintWriter (File file)</code>	Конструктор создает объект <code>PrintWriter</code> для указанного объекта <code>File</code> .
<code>PrintWriter (String filename)</code>	Конструктор создает объект <code>PrintWriter</code> для указанного в строке имени файла.
<code>void print(String s)</code>	Записывает строку в файл.
<code>void print(char c)</code>	Записывает символ в файл.
<code>void print(char[] cArray)</code>	Записывает массив символов в файл.
<code>void print(int i)</code>	Записывает целочисленное значение в файл.
<code>void print(long l)</code>	Записывает значение <code>long</code> в файл.
<code>void print(float f)</code>	Записывает значение <code>float</code> в файл.
<code>void print(double d)</code>	Записывает значения <code>double</code> в файл.
<code>void print(boolean b)</code>	Записывает логическое значение в файл.
Также содержит перегруженные варианты методов <code>println()</code> и <code>printf()</code> .	

Применения методов класса `PrintWriter` для записи данных в текстовый файл иллюстрирует код в примере 7.12.

Пример 7.12. *Запись данных в текстовый файл с помощью методов класса `PrintWriter`.*

Объяснение: в программе создается экземпляр класса `PrintWriter` и записывают две строки в файл `scores.txt`. Каждая строка состоит из имени и фамилии студента (строка) и оценки (целое число).

Вызов конструктора `PrintWriter` создаст новый файл, в том случае если файл не существует. Если же файл уже существует, то на экране появится соответствующее сообщение и программа завершится.

Вызов конструктора `PrintWriter` может выбрасывать исключение ввода/вывода, поэтому в заголовке главного метода `main()` записан код для работы с этим типом исключения: `IOException`. Запись данных в текстовый файл осуществляется методами `print()` и `println()` класса `PrintWriter`

Для закрытия файла обязательным является использование метода `close()`. Если данный метод не вызвать, данные могут быть сохранены в файле не полностью.

```
import java.io.*;
```

```

public class WriteData {
public static void main(String[] args) throws IOException {
    File file = new File("c://scores.txt");
    if (file.exists()) {
        System.out.println("File already exists");
        System.exit(1);
    }
    PrintWriter output = new PrintWriter(file);
    output.print("John Smith ");
    output.println(90);
    output.print("Eric Jones ");
    output.println(85);
    output.close();
}
}

```

John Smith 90
 Eric Jones 85

 scores.txt

Чтение данных из текстового файла с помощью класса Scanner.
 Класс `java.util.Scanner` уже активно используется нами для чтения строк и примитивных значений при вводе с консоли. `Scanner` считывает лексемы, разделенные символами пробела. Для чтения с клавиатуры применяется конструктор класса `Scanner` с входным параметром `System.in`:

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

Для чтения из файла, вызывают конструктор `Scanner`, в который в качестве параметра передается файл:

```
Scanner input = new Scanner(new File(filename));
```

Наиболее часто используемые методы класса `Scanner` приведены в табл.7.14.

Таблица 7.14 – Конструкторы и методы класса `java.util.Scanner`

Конструктор / Метод	Описание
<code>Scanner (File source)</code>	Создает сканер, который производит чтение данных из указанного файла.
<code>Scanner(String source)</code>	Создает сканер, который производит чтение данных из заданной строки.
<code>close()</code>	Закрывает поток сканера.
<code>boolean hasNext()</code>	Возвращает <code>true</code> , если сканер имеет еще данные для чтения, в противном случае возвращается <code>false</code> .
<code>boolean hasNext*()</code>	Возвращает <code>true</code> , если следующая лексема сканера имеет тип, указанный под звездочкой,

	в противном случае возвращается false.
String next()	Возвращает следующую лексему в виде строки из сканера. При этом можно использовать запись <code>System.out.println(scan.next())</code> , если следующего элемента нет то, то бросается исключение <code>java.util.NoSuchElementException</code> .
String nextLine()	Возвращает строку, которая заканчивается разделителем из сканера.
byte nextByte()	Возвращает из сканера следующее слово, преобразованное к типу <code>byte</code> .
short nextShort()	Возвращает из сканера следующее слово, преобразованное к типу <code>short</code> .
int nextInt()	Возвращает из сканера следующее слово, преобразованное к типу <code>int</code> .
long nextLong()	Возвращает из сканера следующее слово, преобразованное к типу <code>long</code> .
float nextFloat()	Возвращает из сканера следующее слово, преобразованное к типу <code>float</code> .
double nextDouble()	Возвращает из сканера следующее слово, преобразованное к типу <code>double</code> .
Scanner useDelimiter(String pattern)	Устанавливает разделитель для сканера и возвращает этот сканер. В качестве аргумента может принимать <code>Pattern</code> .
Scanner findInLine(String pattern)	Принимает в качестве аргумента <code>String</code> или <code>Pattern</code> . Используется для поиска в лексеме того или иного значения.
Scanner skip(String pattern)	Принимает в качестве аргумента <code>String</code> или <code>Pattern</code> . Используется для пропуска лексем.

Программный код в примере 7.13 иллюстрирует применение методов класса `Scanner` для чтения данных из текстового файла `scores.txt`.

Пример 7.13. Применение методов класса *Scanner* для чтения данных из текстового файла.

Объяснение: в программе для считывания данных из файла, вызывается конструктор класса `Scanner`, в который передается `file` - экземпляр класса `java.io.File`.

Вызов конструктора `new Scanner(file)` может вызывать исключение ввода/вывода, поэтому в заголовке главного метода объявляется `throws Exception`.

В каждой итерации цикла `while` из текстового файла считываются имя, фамилия и оценка. После цикла входной файл закрывается. Делать это не

обязательно, однако вызов в конце работы с файлом метода `close()` является хорошей практикой и позволяет освободить ресурсы, занимаемые файлом.

```
import java.util.Scanner;
import java.io.*;
public class ReadData {
    public static void main(String[] args) throws Exception {
        File file = new File("e://scores.txt");
        Scanner input = new Scanner(file);
        while (input.hasNext()) {
            String firstName = input.next();
            String lastName = input.next();
            int score = input.nextInt();
            System.out.println(firstName + " " + lastName + " "
                               + score);
            input.close();
        }
    }
}
```

Результат:

```
John Smith 90
Eric Jones 85
```

Методы `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, `nextDouble()` и `next()` позволяют считывать лексемы (слова), разделенные разделителями. По умолчанию, разделителями являются пробелы. Можно использовать метод `useDelimiter(String regex)`, чтобы установить новый шаблон для разделителей. При чтении лексема автоматически преобразуется в значение `byte`, `short`, `int`, `long`, `float` и `double` для `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()` и `nextDouble()`, соответственно. При выполнении метода `next()` никакого преобразования не выполняется. Если лексема не соответствует ожидаемому типу, то бросается исключение `java.util.InputMismatchException`.

Оба метода `next()` и `nextLine()` читают строку. Метод `next()` считывает строку ограниченную разделителями, и `nextLine()` читает строку, заканчивающуюся строковым разделителем. Строковый разделитель определяется установленной операционной системой. Это может быть `\r \n` в Windows и `\n` в UNIX. Чтобы узнать строковый разделитель на конкретной платформе, используйте `String lineSeparator = System.getProperty("line.separator");`. Если входные данные вводятся клавиатуры, то ввод строки заканчивается нажатием клавиши Enter, которая соответствует символу `\n`.

Рассмотренные выше методы для чтения данных примитивных типов не читают разделитель после лексемы. Если после метода читающего данные примитивных типов вызывается метод `nextLine()`, то он

считывает символы, которые начинаются с данного разделителя и заканчиваются строковым разделителем. Строковый разделитель считывается, но не является частью строки, возвращаемой методом `nextLine()`.

Предположим, текстовый файл с именем `test.txt` содержит строку
34 567

После того, как будет выполнен следующий код,

```
Scanner input = new Scanner(new File("test.txt"));
int intValue = input.nextInt();
String line = input.nextLine();
```

`intValue` содержит 34, а строка `line` содержит символы " ", 5, 6, и 7.

Что произойдет, если ввод осуществляется с клавиатуры? Предположим, вы ввели 34 и нажали клавишу Enter, затем ввели 567 и нажали клавишу Enter, то после выполнения следующего кода:

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
String line = input.nextLine();
```

в переменной `intValue` будет значение 34, а в `line` – пустая строка. Метод `nextInt()` считает 34 и остановится на разделителе, который в данном случае является строковым разделителем (клавиша Enter). Метод `nextLine()` закончится после чтения строкового разделителя и вернет строку, прочитанную перед строковым разделителем, однако до строкового разделителя нет ни одного символа, поэтому строка `line` пустая.

С помощью класса `Scanner` можно прочитать данные из файла или с клавиатуры. Можно также сканировать данные из строки. Например, следующий код

```
Scanner input = new Scanner("13 14");
int sum = input.nextInt() + input.nextInt();
System.out.println("Sum = " + sum);
```

напечатает `Sum = 27`

Рассмотрим еще несколько примеров чтения данных из строки.

Допустим, имеется текстовый файл `in.txt` с таким содержанием:

```
Hello 24 world 87
```

Создадим сканнер нашего файла:

```
Scanner scan = new Scanner(new File("in.txt"));
```

Попробуем вывести всё содержимое файла:

```
while (scan.hasNext()) {
    System.out.println(scan.next());
}
```

Будет выведено:

```
Hello
24
world
87
```

Выведем только числа. Для этого добавим в код блок `if else`:

```
while (scan.hasNext()) {
    if(scan.hasNextBigInteger()) {
        System.out.println(scan.next()); }else{scan.next();}
    }
}
```

На выводе получим:

24

87

Попробуем в файл добавить разделитель / :

```
Scanner scan = new Scanner(new FileReader("in"));
scan.useDelimiter("/");
while (scan.hasNext()) {
    System.out.println(scan.next());
}
```

И изменим файл in.txt:

Hello/24/world 87

Запустим программу и на выводе получим:

Hello

24

world 87

Воспользуемся методом `findInLine()` для того чтобы найти все слова. Воспользуемся регулярным выражением:

```
Pattern patt = Pattern.compile("[a-z A-Z]+");
```

изменим код следующим образом:

```
while (scan.hasNext()) {
    System.out.println(scan.findInLine(patt));
    scan.next();
}
```

На выводе получим:

Hello

world

Если добавить перед циклом:

```
scan.skip("Hello");
```

То на выходе будет только:

world

Задача 7.5. *Напишите программу, заменяющую в текстовом файле все вхождения строки на новую строку. Имя файла и строки передаются из командной строки. Аргументы следующие:*

```
SourceFile TargetFile oldString NewString.
```

Например, при вводе

```
FormatString.java t.txt StringBuilder StringBuffer
```

следует заменить все вхождения StringBuilder на StringBuffer в файле FormatString.java и сохранить новый файл t.txt.

Объяснение: первоначально программа проверяет количество аргументов, переданных в главный метод из командной строки. Если аргументы не переданы программа завершает свою работу и выдает

соответствующее сообщение. Далее проводятся проверки, существуют ли исходный и целевой файлы.

Создается Scanner для исходного файла, и PrintWriter для записи в целевой файл. В цикле while считывается строка из исходного файла и заменяется на новую, далее новая строка записывается в целевой файл. Чтобы гарантировать, что данные будут сохранены в файле должным образом, в конце программы с помощью метода close() закрываем входной и выходной потоки.

В нормальной ситуации, программа завершается после копирования файла. Аварийно программа может завершиться в случае, если не были введены аргументы командной строки, если исходный файл не существует или если целевой файл уже существует.

```
import java.io.*;
import java.util.*;
public class ReplaceText {
    public static void main(String[] args) throws Exception {
        if (args.length != 4) {
            System.out.println("Enter the command-line arguments");
            System.exit(1);
        }

        //Проверяем существует ли исходный файл
        File sourceFile = new File(args[0]);
        if(!sourceFile.exists()){
            System.out.println("Source file " + args[0]
                               + " does not exist");
            System.exit(2);
        }
        //Проверяем существует ли целевой файл
        File targetFile = new File(args[1]);
        if(targetFile.exists()){
            System.out.println("Target file " + args[1]
                               + " already exists");
            System.exit(3);
        }
        //Создаем сканер для ввода и PrintWriter для вывода
        Scanner input = new Scanner(sourceFile);
        PrintWriter output = new PrintWriter(targetFile);
        while(input.hasNext()){
            String s1 = input.nextLine();
            String s2 = s1.replaceAll(args[2], args[3]);
            output.println(s2);
        }
        input.close();
        output.close();
    }
}
```

Упражнения к главе 7

1. Пусть четыре строки `s1`, `s2`, `s3` и `s4` заданы следующим образом:

```
String s1 = "Welcome to Java";  
String s2 = s1;  
String s3 = new String("Welcome to Java");  
String s4 = "Welcome to Java";
```

Каковы результаты следующих выражений?

- 1.1) `s1 == s2`
- 1.2) `s2 == s3`
- 1.3) `s1.equals(s2)`
- 1.4) `s2.equals(s3)`
- 1.5) `s1.compareTo(s2)`
- 1.6) `s2.compareTo(s3)`
- 1.7) `s1 == s4`
- 1.8) `s1.charAt(0)`
- 1.9) `s1.indexOf('j')`
- 1.10) `s1.indexOf("to")`
- 1.11) `s1.lastIndexOf('a')`
- 1.12) `s1.lastIndexOf("o", 15)`
- 1.13) `s1.length()`
- 1.14) `s1.substring(5)`
- 1.15) `s1.substring(5, 11)`
- 1.16) `s1.startsWith("Wel")`
- 1.17) `s1.endsWith("Java")`
- 1.18) `s1.toLowerCase()`
- 1.19) `s1.toUpperCase()`
- 1.20) `"Welcome ".trim()`
- 1.21) `s1.replace('o', 'T')`
- 1.22) `s1.replaceAll("o", "T")`
- 1.23) `s1.replaceFirst("o", "T")`
- 1.24) `s1.toCharArray()`

2. Предположим, что `s1` и `s2` – две строки. Какие из следующих утверждений или выражений являются неверными?

```
String s = new String("new string");  
String s3 = s1 + s2;  
String s3 = s1 - s2;  
s1 == s2;  
s1 >= s2;  
s1.compareTo(s2);  
int i = s1.length();  
char c = s1(0);  
char c = s1.charAt(s1.length()/2);  
char c = s1.charAt(s1.length());
```

3. Что будет напечатано в результате выполнения следующего кода?

```
String s1 = "Welcome to Java";  
String s2 = s1.replace("o", "abc");  
System.out.println(s1);  
System.out.println(s2);
```

4. Пусть строка `s1` равна "Welcome", а `s2` равна "welcome". Запишите коды для следующих заданий:

- 4.1) Проверьте, равны ли `s1` и `s2`, результат присвойте логической переменной `isEqual`;
- 4.2) Проверьте, равны ли `s1` и `s2` без учета регистра, результат присвойте логической переменной `isEqual`;
- 4.3) Сравните `s1` и `s2`, результат присвойте целочисленной переменной `x`.
- 4.4) Сравнивает `s1` и `s2` без учета регистра, результат присвойте целочисленной переменной `x`.
- 4.5) Проверьте, имеет ли `s1` префикс AAA, результат присвойте логической переменной `b`;
- 4.6) Проверьте, имеет ли `s1` суффикс AAA, результат присвойте логической переменной `b`;
- 4.7) Длину `s1` присвойте целочисленной переменной `x`;
- 4.8) Присвойте первый символ `s1` символьной переменной `x`;
- 4.9) Создайте новую строку `s3`, которая объединяет `s1` и `s2`.
- 4.10) Создайте подстроку из `s1`, начиная с индекса 1;
- 4.11) Создайте подстроку из `s1` от индекса 1 до индекса 4;
- 4.12) Создайте новую строку `s3`, преобразовав строку `s1` в нижний регистр;
- 4.13) Создайте новую строку `s3`, преобразовав строку `s1` в верхний регистр;
- 4.14) Создайте новую строку `s3`, удалив пробелы на обоих концах `s1`;

- 4.15) Замените все вхождения символа е на Е в s1, результат запишите в новую строку s3.
- 4.16) Разделите на слова строку Welcome to Java and HTML в массив tokens. В качестве разделителя использовать пробел.
- 4.17) Присвойте индекс первого вхождения символа е в s1 целочисленной переменной x;
- 4.18) Присвойте индекс последнего вхождения строки abc в s1 целочисленной переменной x.

5. Что будет распечатано в результате выполнения следующего кода и почему?

```
public class Test {  
    String text;  
    public Test(String s) {  
        text = s;  
    }  
    public static void main(String[] args) {  
        Test test = new Test("ABC");  
        System.out.println(test);  
    }  
}
```

6. Что будет распечатано в результате выполнения следующего кода?

```
public class Test {  
    public static void main(String[] args) {  
        String s = "Hi, Good Morning";  
        System.out.println(m(s));  
    }  
    public static int m(String s) {  
        int count = 0;  
        for (int i = 0; i < s.length(); i++)  
            if (Character.toUpperCase(s.charAt(i)))  
                count++;  
        return count;  
    }  
}
```

7. Предположим, что строки s1 и s2 определены следующим образом:

```
StringBuilder s1 = новая StringBuilder ("Java");  
StringBuilder s2 = новая StringBuilder ("HTML");
```

Чему будет равна строка s1 после каждого из следующих операторов. Предположим, что операторы выполняются независимо друг от друга.

- 7.1) s1.append(" is fun");
- 7.2) s1.append(s2);
- 7.3) s1.insert(2, "is fun");

```

7.4) s1.insert(1, s2);
7.5) s1.charAt(2);
7.6) s1.length();
7.7) s1.deleteCharAt(3);
7.8) s1.delete(1, 3);
7.9) s1.reverse();
7.10) s1.replace(1, 3, "Computer");
7.11) s1.substring(1, 3);
7.12) s1.substring(2);

```

8. Что будет распечатано в результате выполнения следующего кода?

```

public class Test {
    public static void main(String[] args) {
        String s = "Java";
        StringBuilder builder = new StringBuilder(s);
        change(s, builder);
        System.out.println(s);
        System.out.println(builder);
    }

    private static void change(String s, StringBuilder builder) {
        s = s + " and HTML";
        builder.append(" and HTML");
    }
}

```

9. Определите результаты выполнения следующей программы, при вызове ее следующими тремя способами:

1. java Test I have a dream
2. java Test "1 2 3"
3. java Test

```

public class Test {
    public static void main(String[] args) {
        System.out.println("Number of strings is " + args.length);
        for (int i = 0; i < args.length; i++)
            System.out.println(args[i]);
    }
}

```

10. Напишите содержимое файла temp.txt после выполнения следующей программы:

```

import java.io.*;
public class Test {
    public static void main(String[] args) throws Exception {
        PrintWriter output = new PrintWriter("temp.txt");
        output.printf("amount is %f %e\r\n", 32.32, 32.32);
        output.printf("amount is %5.4f %5.4e\r\n", 32.32, 32.32);
    }
}

```

```
output.printf("%6b\r\n", (1 > 2));
output.printf("%6s\r\n", "Java");
output.close();
}
}
```

11. Предположим, с клавиатуры были введены 45 57,8 789, а затем нажата клавиша Enter. Чему будут равны переменные после выполнения следующего кода:

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
double doubleValue = input.nextDouble();
String line = input.nextLine();
```

12. Предположим, что с клавиатуры ввели 45, нажали клавишу Enter, 57,8, нажали клавишу Enter, 789, и нажали клавишу Enter. Чему будут равны переменные после выполнения следующего кода:

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
double doubleValue = input.nextDouble();
String line = input.nextLine();
```

14. Как известно, можно проверить, является ли строка подстрокой другой строки с помощью метода `indexOf()` класса `String`. Разработайте собственный метод, выполняющий данную функцию. Напишите программу, которая предлагает пользователю ввести две строки, и проверяет, является ли первая строка подстрокой второй.

15. Напишите метод, который подсчитывает количество букв в строке (не используя метод `length`) со следующим заголовком:

```
public static int countLetters(String s)
```

Напишите тестовую программу, которая предлагает пользователю ввести строку и печатает количество букв в строке.

16. Некоторые веб-сайты накладывают определенные правила на вводимые пользователями пароли. Напишите метод, который проверяет, является ли строка допустимым паролем. Предположим, что пароль должен соответствовать следующим правилам:

- содержать не менее восьми символов;
- состоять только из букв и цифр;
- содержать не менее двух цифр.

Напишите программу, которая предлагает пользователю ввести пароль и отображает `Valid Password`, если правила соблюдены или `Invalid Password` в противном случае.

17. Напишите метод, который подсчитывает количества каждой цифры в строке, используя следующий заголовок:

```
public static int[] count(String s)
```

Метод подсчитывает, сколько раз цифра появляется в строке и возвращает массив из десяти элементов, каждый из которых содержит значение счетчика для соответствующей цифры. Так, например, после выполнения `int[] counts = count("12203AB3")`, `counts[0]` равен 1, `counts[1]` равен 1, `counts[2]` равен 2, `counts[3]` равен 2.

Напишите тестовую программу, которая предлагает пользователю ввести строку и отображает количества вхождений каждой цифры в строку.

18. Напишите метод, который возвращает отсортированную строку и имеет следующий заголовок:

```
public static String sort(String s)
```

Например, `sort("acb")` вернет `abc`. Напишите тестовую программу, которая предлагает пользователю ввести строку и выводит отсортированную строку.

19. Напишите метод, который проверяет, является ли два слова анаграммой. Два слова являются анаграммой, если они содержат те же буквы в любом порядке. Так, например, слова `крона` и `нора` являются анаграммой. Заголовок метода:

```
public static boolean isAnagram(String s1, String s2)
```

Напишите тестовую программу, которая предлагает пользователю ввести две строки и, если они анаграммы, отображает `two strings are anagrams` либо `two strings are not anagrams` в противном случае.

20. Напишите программу, которая считывает из текстового файла строку и выводит на экран количество заглавных букв в данной строке.

21. Напишите рекуррентный метод для удаления непустого каталога на диске.

8 ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Массивы, рассмотренные в главе 4, являются самой распространенной *статической структурой данных*, поддерживаемой в большинстве языков программирования. На их основе могут формироваться более сложные структуры. Однако у массивов есть недостатки. Мы уже отмечали, что массивы имеют фиксированный размер, определяемый при их создании, который не может изменяться во время выполнения программы. Кроме того в массивах медленно работают такие операции как поиск и вставка, так как приходится в среднем перемещать половину элементов массива. Однако существует много задач, где нужны более гибкие структуры данных. Для таких задач характерно, что не только значения, но и структура переменных меняется во время вычисления. Поэтому их называют *динамическими структурами данных*. К таким структурам относятся стеки, очереди, связанные списки, деревья. Они не входят в синтаксис языка, но широко представлены в библиотеках JDK. Это так называемые классы-коллекции, содержащиеся в пакете `java.util`. О том, как правильно работать с коллекциями вы можете узнать из документации Java, а в данной главе попробуем подробнее рассмотреть сами эти структуры и их реализацию.

8.1 Пример реализации динамического массива

Проблема массива, связанная с фиксированной длиной, может быть решена. Один из вариантов – создать собственный класс, задающий массив с «плавающей» верхней границей, в котором описать методы для динамического добавления и удаления элементов. Например, если операция вставки должна привести к переполнению массива, то соответствующий метод создает новый массив большего размера, копирует в него содержимое старого массива, а затем вставляет новый элемент. Несмотря на то, что стандартная библиотека Java имеет определение подобного класса – `java.util.ArrayList`, попробуем привести простой пример собственной реализации такого «динамического массива».

Пример 8.1 Определим класс *DynamicArray*, задающий массив с поддержкой динамического расширения и содержащий методы вставки и удаления элементов.

Объяснение: методы `add()` и `remove()` построены в соответствии с алгоритмами вставки и удаления элементов массива, рассмотренными в главе 4. Каждый раз при вставке нового элемента создается временный массив `tmp[]`, размер которого на единицу больше размера текущего массива `data[]`. Соответственно при удалении – размер временного

массиву на единицу меньше размера массива data[]. Для перемещения элементов влево и вправо используется метод arraycopy() класса System.

```
import static java.lang.System;
public class DynamicArray {
    int[] data = {};

    //метод вставки элемента в конец массива
    void add(int elem) {
        add(data.length, elem);
    }
    //метод удаления элемента в конце массива
    void remove() {
        remove(data.length - 1);
    }
    //метод вставки элемента в заданную позицию
    void add(int index, int elem) {
        int[] tmp = new int[data.length+1];
        arraycopy(data, 0, tmp, 0, index);
        arraycopy(data, index, tmp, index+1, data.length-index);
        tmp[index] = elem;
        this.data = tmp;
    }
    //метод удаления элемента в заданной позиции
    void remove(int index) {
        int[] tmp = new int[data.length-1];
        arraycopy(data, 0, tmp, 0, index);
        arraycopy(data, index+1, tmp, index, data.length- index-1);
        this.data = tmp;
    }
}
```

Выполним тестирование созданного класса DynArray. Для этого создадим новый класс DynamicArrayTest:

```
import java.util.*;
class DynamicArrayTest {
    public static void main(String[] args) {
        DynamicArray dArr = new DynamicArray();
        //массив пока пуст
        System.out.println(Arrays.toString(dArr.data));
        //добавляем в массив нечетные числа: 1,3,5,7,9
        for(int i=1; i<10; i+=2)
            dArr.add(i);
        System.out.println(Arrays.toString(dArr.data));
        //удаляем последний элемент
        dArr.remove();
        System.out.println(Arrays.toString(dArr.data));
        //удаляем элемент с индексом 2
        dArr.remove(2);
        System.out.println(Arrays.toString(dArr.data));
    }
}
```

```

        //добавляем элемент 2 в позицию 1
        dArr.add(1, 2);
        System.out.println(Arrays.toString(dArr.data));
    }
}

```

Результат:

```

[]
[1, 3, 5, 7, 9]
[1, 3, 5, 7]
[1, 3, 7]
[1, 2, 3, 7]

```

Класс `DynamicArray` хотя и решает поставленную перед ним задачу, но является не эффективным и тяжеловесным, так как каждый раз при выполнении операций вставки и удаления элементов происходит перераспределение памяти. При работе с такой структурой приходится каждый раз копировать память, объем которой прямо пропорционален размеру массива.

8.2 Абстрактные типы данных

Типы данных – это множество значений и множество операций над этими значениями. Например, значения примитивного типа данных `int` представляют собой целые числа от -2^{31} до $2^{31}-1$, а к операциям над ними относятся $+$, $*$, $-$, $/$, $\%$, $<$ и $>$. Однако, кроме примитивных типов данных в Java существуют еще и ссылочные типы. Работая с примитивными типами можно писать программы, применение которых ограничивается только обработкой чисел, тогда как ссылочные типы позволяют писать программы, работающие со строками, изображениями, звуками и сотнями других абстракций. Таким образом, программист может определять *собственные типы данных* для реализации любой необходимой абстракции. При этом приходится заботиться о способах представления данных и об операциях над ними (для ссылочного типа действия над данными выполняются с помощью методов). Конечно, часто именно способ представления определяет возможные методы, но иногда можно ввести методы независимо от представления данных, и рассматривать различные способы реализации этих методов для различных представлений структуры данных.

Когда хотят отделить понятие об операциях от конкретного представления тех или иных данных, говорят, что задается абстрактный тип данных. *Абстрактный тип данных* (АТД) можно задать, если перечислить все операции, разрешенные над значениями этого типа, и определить правила, которым должны удовлетворять эти операции.

Как правило, методы разрабатываемых библиотек принято описывать в *интерфейсах прикладного программирования* (API–интерфейсах), которые содержат имя библиотеки и сигнатуры, а также краткие описания всех задействованных методов. В следующих разделах текущей главы попробуем создать несколько простых библиотек для работы с абстрактными типами данных: стеком и очередью. Для описания поведения абстрактного типа данных будем использовать API – интерфейс, который представляет собой список конструкторов и методов экземпляров (операций) с неформальным описанием действия каждого из них. Назовем *клиентом* программу, которая вызывает какой-то метод из другой библиотеки, и *реализацией* – Java-код, который реализует методы из API – интерфейса. Так, в примере 8.1 созданный нами класс `DynamicArray` является реализацией, а тестирующий класс `DynamicArrayTest` – клиентом. Причем для рассматриваемого примера АДТ API–интерфейс мог бы иметь следующий вид:

```
public class DynamicArray
```

<code>void add(int elem)</code>	метод вставки элемента в конец массива
<code>void remove()</code>	метод удаления элемента в конце массива
<code>void add(int index, int elem)</code>	метод вставки элемента в заданную позицию
<code>void remove(int index)</code>	метод удаления элемента в заданной позиции

Основное назначение API–интерфейса состоит в возможности *отделения* клиента от реализации: клиент ничего не должен знать о реализации, кроме информации, приведенной в API–интерфейсе, а реализация не должна учитывать особенности конкретного клиента. API–интерфейсы позволяют отдельно разработать код для произвольных целей, а затем использовать его сколько угодно. В связи с этим программисты обычно рассматривают API–интерфейс как *соглашение (контракт)* между клиентом и реализацией, т.е. ясное описание того, что должен делать каждый метод. Наша цель при реализации – соблюдать условия этого контракта.

8.3 Стеки

Стек представляет собой контейнер, работа с которым – добавление и удаление – осуществляется по принципу LIFO (last-in first-out – последним пришел – первым ушел). Добавление объектов в стек может осуществляться в любой момент, удаляется же лишь объект, добавленный последним (рис. 8.1).

Стек LIFO – это абстрактный тип данных, который поддерживает следующий API интерфейс:

public class **Stack**<Item> implements Iterable<Item>

Stack()	создание пустого стека
void push(Item item)	добавление элемента в вершину стека
Item pop()	удаление элемента из вершины стека
boolean isEmpty()	возвращает логическое значение, подтверждающее, что стек пуст
int size()	количество элементов в стеке

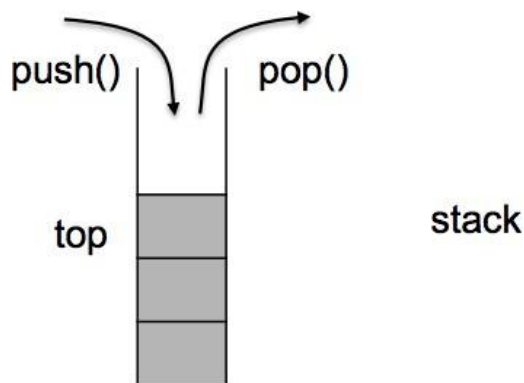


Рис.8.1 – Операции со стеком

8.3.1 Стек фиксированной емкости

В разделе 5.9 уже была приведена простая реализация стека на основе массива фиксированной длины. В данном разделе попробуем усовершенствовать старую реализацию.

Внесем некоторые изменения в представление стека. В качестве переменных экземпляра будут использоваться массив $s[]$, который содержит значения стека и целое число N , содержащее количество элементов в стеке. Для удаления (извлечения) элемента нужно уменьшить на единицу N и вернуть $s[N]$; для вставки (добавления) нового элемента нужно занести этот новый элемент в $s[N]$, а затем увеличить на единицу N . Эти операции обладают следующими свойствами:

- Элементы находятся в массиве в порядке их вставки;
- Стек пуст, когда $N = 0$;

Вершина (непустого) стека находится в элементе $s[N-1]$.

Один из недостатков реализации типа **Stack** в разделе 5.9, состоит в том, что он может работать только с элементами типа **int**. Если

понадобится стек значений `double` или `String`, то придется разрабатывать другой класс с похожим кодом и везде заменить `int` на `double`, потом возможно на `String` или любой другой тип. Для устранения подобных проблем в Java была введена базовая концепция параметризованных типов – *обобщений* (*generics*). Это дополнительный механизм языка программирования, который позволяет разрабатывать более ясный и компактный код.

В примере 8.2 приводится реализация обобщенного стека, в котором все вхождения `int` заменены на `Item`, а объявление класса в первой строке имеет вид: `class Stack <Item>`. Идентификатор `Item` – это *параметр типа*, т.е. заменитель некоторого конкретного типа, который потом будет использовать клиент. Запись `Stack<Item>` можно понимать как «стек элементов». При реализации типа `Stack` мы не знаем конкретный тип `Item`, но клиент может использовать наш стек для любого типа данных, в том числе и определенного через какое-то продолжительное время после разработки реализации. При создании стека клиентский код сообщает конкретный тип: идентификатор `Item` можно заменить именем *любого* ссылочного типа данных (в любом месте, где он может появиться). Например, можно записать код для применения стека с объектами `String`:

```
Stack<String> stack = new Stack<String>();
stack.push("teast");
...
String next = stack.pop();
```

При попытке добавить объект любого другого типа, отличного от `String` возникнет ошибка времени компиляции (*runtime exception*). Без обобщений пришлось бы определять (и реализовывать) различные API-интерфейсы для каждого типа данных, который может понадобиться в коллекциях, а обобщения позволяют ограничиться одним API-интерфейсом (и одной реализацией) для всех типов данных, даже таких типов, которые еще только будут реализованы.

На место параметров типа должны подставляться ссылочные типы, но в Java имеются специальные механизмы, которые позволяют использовать обобщенный код и с примитивными типами. Так в Java автоматически осуществляется преобразования примитивных типов в соответствующие им ссылочные типы: `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long` и `Short` в присваиваниях, аргументах методов и арифметических или логических выражениях. Например, как в следующем коде:

```
Stack<Integer> stack = new Stack<Integer>();
stack.push(15);           //автоупаковка (int -> Integer)
```

```
int i = stack.pop();    //автораспаковка (Integer -> int)
```

Автоматическое приведение типа к типу оболочки называется *автоупаковкой*, а автоматическое приведение типа оболочки к примитивному – *автораспаковкой*. В примере Java автоматически выполняет приведение (автоупаковку) примитивного значения 15 к типу Integer при передаче его в метод push(). Метод pop() возвращает значение Integer, который Java приводит (распаковывает) к типу int перед его присваиванием переменной i.

Пример 8.2. Реализация абстрактного типа данных для обобщенного стека фиксированной длины.

Обратите внимание на то, что *обобщенные массивы в Java не разрешены*, поэтому в конструкторе Stack вместо записи `s = new Item[size]` выполнено приведение вида: `s = (Item[]) new Object [size]`.

```
public class Stack <Item>{
    private Item[] s;
    private int N;

    public Stack(int size){
        s = (Item[]) new Object [size];
    }

    public int size() {
        return N;
    }
    public boolean isEmpty(){
        return N==0;
    }
    public void push(Item item){
        s[N++] = item;
    }
    public Item pop(){
        return s[--N];
    }
}
```

8.3.2 Стек с переменным размером массива

Выбор массива для хранения содержимого стека в примере 8.2 означает, что клиенты должны заранее оценивать максимальный размер

стека. Java не допускает изменение размера массива после его создания, поэтому стек всегда использует объем памяти, пропорциональный этому максимуму. Если размер коллекции превысит размер массива, клиент рискует столкнуться с *переполнением*.

– Добавим в реализацию стека динамическое изменение размера массива `s[]`, чтобы массив был достаточно большим для хранения всех элементов, но не слишком большим, чтобы не тратить зря большой объем памяти.

Для реализации стека с переменным размером массива вначале реализуем метод, который перемещает стек в массив другого размера:

```
private void resize (int max)
{ //Перенос стека размером N <= max в новый массив размером max
  Item[] temp = (Item[]) new Object[max];
  for(int i =0; i<N; i++)
    temp[i] = s[i];
  s = temp;
}
```

Теперь в метод `push()` следует добавить проверку, исчерпана ли емкость массива. Определить это можно сравнив размер стека `N` с размером массива `s.length`. Если места уже нет, *увеличим вдвое* размер массива, а потом, как и раньше, вставим новый элемент с помощью операции `s[N++] = item`:

```
public void push (Item item)
{ // Добавление элемента на верхушку стека
  if (N==s.length) resize (2* s.length);
  s[N++] = item;
}
```

Аналогично и в методе `pop()`: сначала удаляем элемент, а затем *уменьшаем вдвое* размер массива, если он слишком велик. Удобно выполнять проверку, меньше ли размер стека, чем *четверть* размера массива. Тогда после уменьшения массива он будет заполнен примерно наполовину и сможет выполнить достаточное количество операций `push()` и `pop()`, прежде чем появится необходимость снова изменить размер массива.

```
public Item pop ()
{ // Удаление элемента с верхушки стека
  Item item = s[--N];
  s[N] = null; //убираем безхозную ссылку
  if (N > 0 && N == s.length/4) resize (s.length/2);
  return item;
}
```

При такой реализации стек никогда не переполнится и никогда не будет заполнен менее чем на четверть (кроме случая пустого стека, когда размер массива равен 1).

Теперь разберем, как будет осуществляться проход по элементам коллекции. Проход должен выполняться во время *итерации* по коллекции и не должен зависеть от деталей ее реализации. Для стека подобный обход должен выполняться в обратном порядке. Как этого можно добиться?

Для итерации по элементам коллекции используют оператор `foreach`. Так, например, выглядит фрагмент клиентского кода, который выводит все элементы коллекции строк, по одной в строке вывода:

```
Stack<String> collection = new Stack<String>();  
...  
for (String s : collection)  
    System.out.println(s);  
...
```

Данный код примерно эквивалентен следующему коду:

```
Iterator<String> i = collection.iterator();  
while (i.hasNext()) {  
    String s = i.next();  
    System.out.println(s);  
}
```

В этом коде уже хорошо видны все ингредиенты необходимые для реализации любой итерируемой коллекции:

- коллекция должна реализовывать метод `iterator()`, который возвращает объект `Iterator`;
- класс `Iterator` должен содержать два метода: `hasNext()` (возвращающий логическое значение) и `next()` (возвращающий обобщенный элемент из коллекции).

Итератор – это объект класса, в котором реализованы методы `hasNext()` и `next()`, как указано в следующем интерфейсе (из библиотеки `java.util.Iterator`):

```
public interface Iterator<Item>  
{  
    boolean hasNext();  
    Item next();  
    void remove();  
}
```

В интерфейсе задан и метод `remove()`, но мы будем указывать для него пустой метод, поскольку совместного использования итерации и операций, изменяющих структуру данных, лучше не допускать.

Чтобы сделать класс итерируемым, необходимо выполнить `import java.util.Iterator` и добавить в начало объявления класса конструкцию

implements Iterable<Item>, которая означает, что этот класс должен соответствовать интерфейсу

```
public interface Iterable<Item>
{
    Iterator<Item> iterator();
}
```

(находится в библиотеке java.lang.Iterable), а затем добавить в класс метод iterator(), возвращающий объект Iterator<Item>.

Создадим свой итератор – вложенный класс ReverseArrayIterator

```
private class ReverseArrayIterator implements Iterator<Item>
{
    private int i = N;
    public Boolean hasNext() {return i > 0; }
    public Item next() {return s[--i]; }
    public void remove() {}
}
```

и добавим в класс такой метод:

```
public Iterator<Item> iterator() {
    return new ReverseArrayIterator();
}
```

Теперь клиент, использующий оператор foreach для этого класса, обретет поведение, эквивалентное обычному циклу for для итерации по элементам массива, но в соответствии с правилом LIFO, и ему не придется вникать в строение этих массивов (детали реализации).

Пример 8.3. *Обобщенная реализация API-интерфейса Stack с возможностью итерации по элементам коллекции. Размер массива изменяется так, чтобы он оставался примерно пропорциональным размеру стека.*

```
import java.util.Iterator;
public class Stack<Item> implements Iterable<Item>
{
    private Item[] s;           // элементы стека
    private int N;              // количество элементов

    // create an empty stack
    public Stack() {
        s = (Item[]) new Object[1];
    }

    public boolean isEmpty() { return N == 0; }
    public int size()         { return N; }
}
```

```

// Перенос стека в новый массив размером max
private void resize(int max) {
    Item[] temp = (Item[]) new Object[max];
    for (int i = 0; i < N; i++) {
        temp[i] = s[i];
    }
    s = temp;
}

// Добавление элемента на верхушку стека
public void push(Item item) {
    if (N == s.length) resize(2*s.length);
    s[N++] = item;
}

// Удаление элемента с верхушки стека
public Item pop() {
    Item item = s[N-1];
    s[N-1] = null;
    N--;
    if (N > 0 && N == s.length/4) resize(s.length/2);
    return item;
}

public Iterator<Item> iterator() {
    return new ReverseArrayIterator();
}

//Поддержка итерации по правилу LIFO
private class ReverseArrayIterator implements Iterator<Item>
{
    private int i;
    public ReverseArrayIterator() {
        i = N;
    }
    public boolean hasNext() {
        return i > 0;
    }
    public void remove() { }
    public Item next() {return s[--i];}
}
}

```

Выполним тестирование созданного класса Stack. Для этого создадим новый класс StackDemo. Клиент вводит текст с консоли и выводит его на экран в обратном порядке.

```
import java.util.*;
```

```

class StackDemo{
public static void main(String[] args) {
    Stack<String> s = new Stack<String>();
    Scanner scanner = new Scanner (System.in);
    String str = scanner.nextLine();
    String item;
    Scanner tokenizer = new Scanner (str);
    while (tokenizer.hasNext())
    {
        item = tokenizer.next();
        s.push(item);
    }
    while (!s.isEmpty())
        System.out.print(s.pop() + " ");
    System.out.println("(" + s.size() + " текущий размер стека)");
}
}

```

Результат:

```

to be or not to be
be to not or be to (0 текущий размер стека)

```

В тестирующем клиенте при необходимости вывода содержимого стека без извлечения элементов, т.е. для организации прохода по коллекции, достаточно строки:

```

while (!s.isEmpty())
    System.out.print(s.pop() + " ");

```

заменить на строки:

```

for(String token : s)
    System.out.print(token+" ");

```

при этом стек останется полным. Элементы просто отображаются из стека на экран, но не извлекаются из него.

8.4 Двухстековый алгоритм Дейкстры для вычисления выражения

Рассмотрим классический пример, который демонстрирует эффективность применения стека для вычисления арифметических выражений, а заодно и пользу обобщений.

Напишем программу, которая принимает на вход строку (выражение) подобное такому $(1 + ((2 + 3) * (4 * 5)))$ и выдает число, равное значению этого выражения. Применим для этого, разработанный в 1960-х годах алгоритм Э.У.Дейкстры, использующий два стека: один для операндов, а другой для операций.

Для простоты примем следующее явное рекурсивное определение: *арифметическое выражение* – это либо число, либо открывающая скобка, за которой следует арифметическое выражение, символ операции, еще одно арифметическое выражение и закрывающая скобка. Алгоритм будем применять только к *полностью скобочным* арифметическим выражениям! Будем поддерживать следующие операции: +, -, *, и /, а также операцию извлечения квадратного корня sqrt, которая принимает только один аргумент. Могут быть добавлены и другие операции. Рассмотрим подробнее сам алгоритм.

Итак, будем выбирать элементы выражения слева направо, и помещать их в стек в соответствии с четырьмя возможными случаями:

- помещать операнды в стек операндов;
- помещать операции в стек операций;
- игнорировать левые скобки;
- при достижении правой скобки извлечь операцию, извлечь необходимое количество операндов и поместить в стек операндов результат действия операции над этими операндами.

После обработки последней скобки в стеке должно остаться только одно значение, которое и является значением выражения (рис.7.2).

Каждый раз, когда алгоритм встречает подвыражение, состоящее из двух операндов с операцией между ними и заключенное в скобки, он оставляет в стеке операндов результат выполнения этой операции над этими операндами. Это действие можно повторять многократно, пока не останется единственное значение:

$(1 + ((2 + 3) * (4 * 5)))$

$(1 + (5 * (4 * 5)))$

$(1 + (5 * 20))$

$(1 + 100)$

101

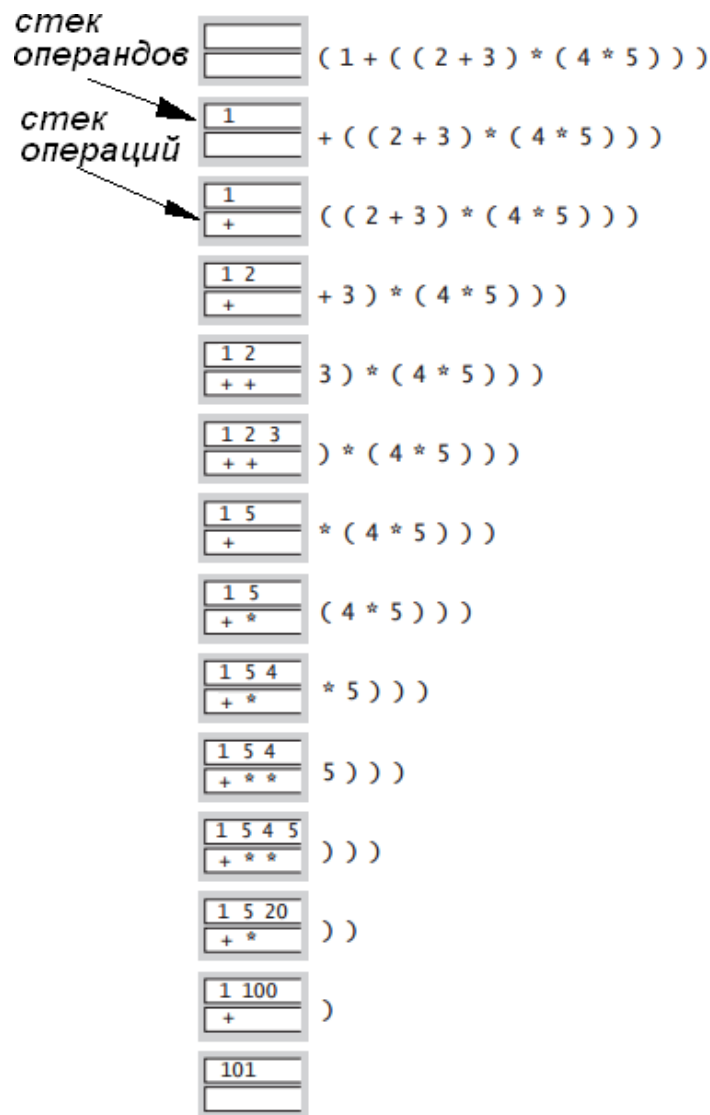


Рис.8.2 – Трассировка двухстекового алгоритма Дейкстры для вычисления арифметического выражения

Пример 8.4. Двухстековый алгоритм Дейкстры для вычисления выражения.

Класс Expression представляет собой простой пример интерпретатора – программы, которая выполняет вычисление, заданное некоторой строкой, и получает результат этого вычисления.

```
import java.util.*;
public class Expression
{
    public static void main (String[] args)
    {
        Stack <String> ops = new Stack<String>(100);
        Stack <Double> vals = new Stack<Double>(100);

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

```

System.out.println("Enter expression:");
String s = null;
String str = scanner.nextLine();
Scanner tokenizer = new Scanner (str);
    while (tokenizer.hasNext())
    {
        s = tokenizer.next();
        if      (s.equals("("))      continue ;
        else if (s.equals("+"))      ops.push(s);
        else if (s.equals("-"))      ops.push(s);
        else if (s.equals("*"))      ops.push(s);
        else if (s.equals("/"))      ops.push(s);
        else if (s.equals("sqrt")) ops.push(s);
        else if (s.equals(""))
        {
            //Если ")", то выталкивание, вычисление и вталкивание результата
            String op = ops.pop();
            double v = vals.pop();
            if      (op.equals("+"))  v = vals.pop() + v;
            else if (op.equals("-"))  v = vals.pop() - v;
            else if (op.equals("*"))  v = vals.pop() * v;
            else if (op.equals("/"))  v = vals.pop() / v;
            else if (op.equals("sqrt")) v = Math.sqrt(v);
            vals.push(v);
        }
        // Элемент не операция и не скобка: вталкиваем значение double
        else vals.push(Double.parseDouble(s));
    }
    System.out.println(vals.pop());
}

```

Результат:

```

Enter expression:
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
Enter expression:
( ( 1 + sqrt ( 5.0 ) ) / 2.0 )
1.618033988749895

```

Приведенный клиент класса Stack использует два стека для вычисления арифметических выражений и демонстрирует важный вычислительный процесс – интерпретацию строки как программы и выполнение этой программы для вычисления нужного результата. Обобщения позволяют использовать единственный класс Stack для реализации одного стека для значений String, а другого – для значений Double. Для простоты здесь предполагается, что значение является полностью скобочными, а числа и символы разделены пробелами.

8.5 Очередь

Очередь FIFO (или просто *очередь*) – это коллекция, в которой действует правило *первым вошел – первым вышел* (first in, first out – FIFO). Такое правило выполнения задач в порядке поступления мы часто наблюдаем в повседневной жизни: люди, ожидающие в очереди или, например, задачи, ожидающие обслуживания приложением в компьютере. В основе данного правила обслуживания лежит принцип честности: тот кто ждал дольше всех, обслуживается первым. Очереди позволяют моделировать многие явления и поэтому играют важную роль во многих приложениях. Обычная причина использования очередей в приложениях – сохранение элементов в коллекции *в относительном порядке их поступления*: они выбираются из очереди в том же порядке, в котором заносятся (рис.8.3).

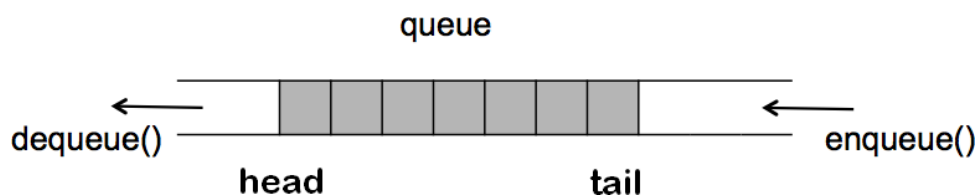


Рис.8.3 – Операции с очередью

Очередь – это абстрактный тип данных, который поддерживает следующий API интерфейс:

<code>public class Queue<Item> implements Iterable<Item></code>	
<code>Queue()</code>	создание пустой очереди
<code>void enqueue(Item item)</code>	добавление элемента
<code>Item dequeue()</code>	удаление самого «старого» элемента
<code>boolean isEmpty()</code>	пуста ли очередь?
<code>int size()</code>	количество элементов в очереди

Реализуем API – интерфейс Queue используя два индекса в виде переменных экземпляров: переменная head для начала очереди и переменная tail для конца очереди:

- head – указатель на индекс ячейки, в которой находится первый элемент очереди (кандидат на удаление при выполнении метода dequeue()) при условии, что очередь не пустая (в этом случае head = tail).

- tail – указатель на хвост (конец очереди), то есть индекс следующей доступной (свободной) ячейки массива.

Чтобы удалить элемент, нужно извлечь его с помощью указателя head, а затем увеличить значение head на единицу для получения индекса

следующей ячейки массива; чтобы вставить элемент, нужно занести его по указателю `tail`, а затем увеличить значение `tail` на единицу для получения следующей доступной ячейки массива.

В программной реализации часто очередь рассматривается как «круговой» массив, который продолжается от `q[0]` до `q[N-1]`, а затем снова возвращается к `q[0]`, т.е. можно обнулять значение `head` и `tail` в случае если в результате их увеличения значения превышают размер очереди. Таким образом, если увеличение индекса вывело его за конец массива, то индекс нужно обнулить.

Можно использовать и другие способы, например, при каждом увеличении значения `head` и `tail` на 1 записывать эту операцию в виде `«(head + 1) % N»` или `«(tail + 1) % N»` соответственно, где `%` обозначает операцию деления по модулю. С помощью операции деления по модулю решается проблема превышения пределов массива.

Пример 8.5. *Обобщенная реализация API-интерфейса `Queue` с возможностью итерации по элементам коллекции. Размер массива изменяется так, чтобы он оставался примерно пропорциональным размеру очереди.*

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class Queue<Item>
    implements Iterable<Item> {
    private Item[] q;      // элементы очереди
    private int N = 0;     // количество элементов в очереди
    private int head = 0;  // индекс первого элемента очереди
    private int tail = 0;  // индекс следующей доступной ячейки

    public Queue() {
        q = (Item[]) new Object[2];
    }

    public boolean isEmpty() { return N == 0; }
    public int size() { return N; }

    private void resize(int max) {
        Item[] temp = (Item[]) new Object[max];
        for (int i = 0; i < N; i++) {
            temp[i] = q[(head + i) % q.length];
        }
        q = temp;
        head = 0;
        tail = N;
    }

    //Вставить элемент в конец очереди
    public void enqueue(Item item) {
```



```

        if (N == q.length) resize(2*q.length);
        q[tail++] = item;
        if (tail == q.length) tail = 0;
        N++;
    }

    //удалить элемент из начала очереди
    public Item dequeue() {
        if (isEmpty())
            throw new RuntimeException("Queue underflow");
        Item item = q[head];
        q[head] = null;
        N--;
        head++;
        if (head == q.length) head = 0;
        if (N > 0 && N == q.length/4) resize(q.length/2);
        return item;
    }

    public Iterator<Item> iterator() {
        return new ArrayIterator(); }

    private class ArrayIterator implements Iterator<Item>
{
    private int i = 0;
    public boolean hasNext() { return i < N;}
    public void remove() { }
    public Item next() {
        if (!hasNext())
            throw new NoSuchElementException();
        Item item = q[(i + head) % q.length];
        i++;
        return item;
    }
}

    }}
    *****Test client*****
import java.util.*;
class QueueDemo{
    public static void main(String[] args) {
        Queue<String> q = new Queue<String>();
        Scanner scanner = new Scanner (System.in);
        String str = scanner.nextLine();
        String item;
        Scanner tokenizer = new Scanner (str);
        while (tokenizer.hasNext())
        {
            item = tokenizer.next();
            q.enqueue(item);
        }
        while (!q.isEmpty())
            System.out.print(q.dequeue() + " ");
    }
}

```

```
        System.out.println("(" + q.size() + " текущий размер  
очереди )");  
    }  
}
```

Результат:

```
to be or not to be  
to be or not to be (0 текущий размер очереди )
```

8.6 Связные списки

Рассмотрим структуру данных связный список, которая удобна для представления данных в реализации АДТ коллекции. Данная структура данных не имеет непосредственной поддержки в языке Java и может быть взята за основу при построении более сложных структур данных.

Итак, *связный список (linked list)* – это рекурсивная структура данных, которая либо пуста (null), либо представляет собой ссылку на *узел (node)*, который содержит обобщенный элемент и ссылку на связный список.

Узел в этом определении – некая абстрактная сущность, которая может содержать произвольные данные и ссылку на другой узел, и которая выполняет ключевую роль в построении связных списков.

Создание связного списка. Запишем *вложенный класс*, который определяет абстракцию узла:

```
private class Node {  
    Item item;  
    Node next;  
}
```

Класс Node содержит две переменных экземпляра типа Item (параметризованный тип) и Node. Как и любой другой тип данных, объект типа Node создается с помощью вызова конструктора new Node() без аргументов. В результате создается ссылка на объект Node, в котором обе переменных экземпляра инициализированы значение null. Переменная item – это заполнитель для любого типа данных, которые понадобилось структурировать в виде связного списка, а переменная типа Node характеризует связную природу структуры данных. Класс Node представляет собой пример рекурсивного объявления типов. Объявлен класс Node, внутри которого объявлено поле типа Node.

Далее на основе рекурсивного определения можно представить связный список переменной типа Node – необходимо просто следить, чтобы ее поле next содержало или null, или ссылку на узел Node, поле next которого является ссылкой на связный список. Например, чтобы создать

связный список, содержащий строки to, be и or, вначале понадобится создать экземпляры Node для каждого элемента:

```
Node first = new Node();  
Node second = new Node();  
Node third = new Node();
```

потом занести в поле item каждого узла нужное значение:

```
first.item = "to";  
second.item = "be";  
third.item = "or";
```

и затем заполнить поля next, так чтобы собрать из отдельных элементов связный список:

```
first.next = second;  
second.next = third;
```

В third.next остается значение null, которое занесено во время создания экземпляра. Переменная third представляет собой связный список, т.к. это ссылка на узел, который содержит ссылку null – т.е. нулевую ссылку на пустой связный список. Переменная second представляет собой связный список, поскольку это ссылка на узел, который содержит ссылку на связный список third. И переменная first также представляет собой связный список, потому что это ссылка на узел, который содержит ссылку на связный список second. Эти операторы присваивания представлены на рис. 8.4.

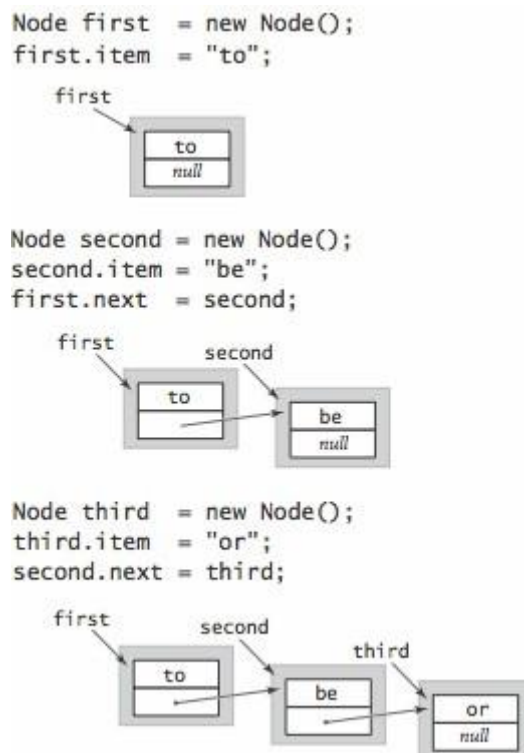


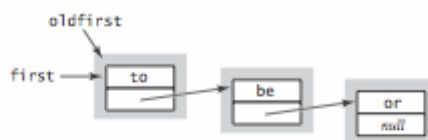
Рис. 8.4 – Сборка списка из элементов

Связный список представляет последовательность элементов. В рассмотренном выше примере переменная `first` представляет последовательность `to be or`. Такую последовательность можно было оформить и в виде массива `String[] s = {"to", "be", "or"}`. Однако в связанных списках легче выполнить вставку элемента в последовательность и удаление элемента из последовательности.

Вставка в начало списка. Для вставки строки `not` в начало списка, в котором первый узел хранит строку `first`, нужно сохранить ссылку `first` в переменной `oldfirst`, занести в `first` ссылку на новый `Node`, и затем занести в поле `item` ссылку на строку `not`, а в поле `next` – значение `oldfirst` (рис. 7.5). Код вставки узла в начало связанного списка требует выполнения лишь нескольких операторов присваивания, поэтому необходимое для этого время не зависит от длины списка.

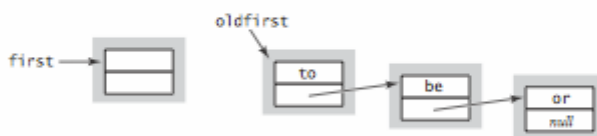
Сохранение ссылки на список

```
Node oldfirst = first;
```



Создание нового головного узла

```
first = new Node();
```



Заполнение переменных экземпляра в новом узле

```
first.item = "not";  
first.next = oldfirst;
```

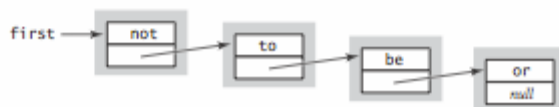


Рис.8.5 – Вставка нового узла в начало связанного списка

Удаление из начала списка. Чтобы удалить первый узел из списка достаточно присвоить переменной `first` значение `first.next` (рис. 7.6). Обычно перед этим значение данного элемента присваивается какой-то переменной типа `Item`, так как после изменения значения `first` доступ к первому узлу может оказаться невозможным. Объект бывшего первого узла становится висячим, и система управления памятью в Java в какой-то момент возвращает занимаемую узлом память в пул доступной памяти.

Эта операция требует выполнения лишь одного оператора присваивания, и поэтому время ее выполнения также не зависит от длины списка.

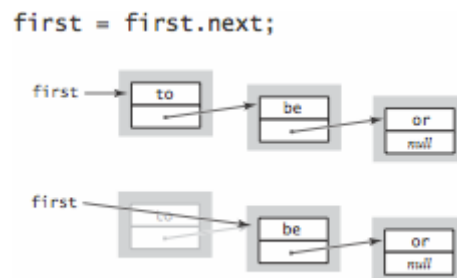
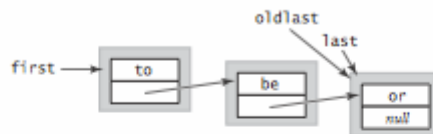


Рис.8.6 – Удаление первого узла из связанного списка

Вставка в конец очереди. Для вставки узла в конец очереди нужно ссылку на последний узел списка изменить так, чтобы она указывала на новый узел со вставленным элементом (рис.8.7).

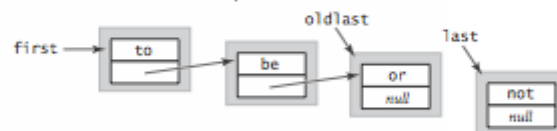
Создание ссылки на последний узел

```
Node oldlast = last;
```



Создание нового конечного узла

```
Node last = new Node();  
last.item = "not";
```



Привязка нового узла в конец списка

```
oldlast.next = last;
```

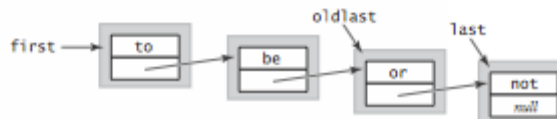


Рис.8.7 – Вставка нового узла в конец связанного списка

Идея структуры состоит в том, что каждый экземпляр класса Node находясь в памяти будет иметь в себе число типа `int`, а втором полем будет ссылочный тип `Node` – ссылка указывающая либо на `null`, либо на другое поле такого же типа `Node`, где тоже будет храниться целое число и ссылка на такую же структуру данных (рис.8.1). Мы получили цепочку объектов, у каждого объекта есть ссылка на следующий.

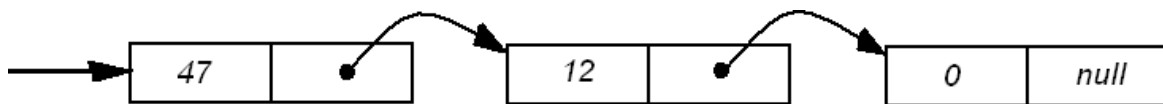
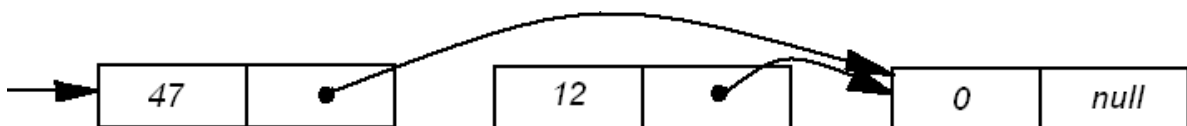


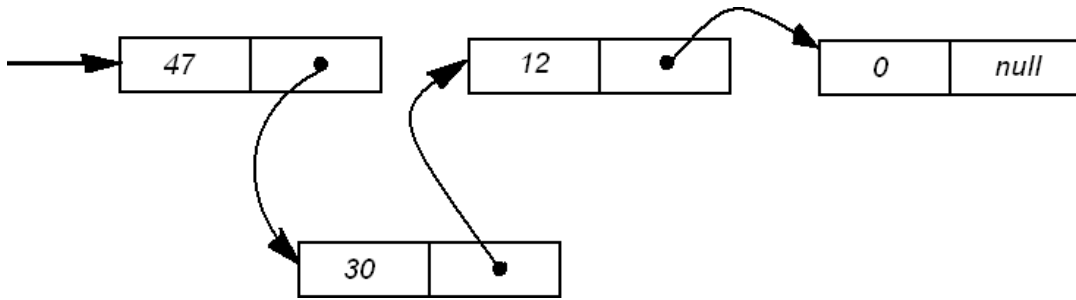
Рис.7.1-

```
public class Node {  
    int value;  
    Node next;  
  
    Node(int value, Node next) {  
        this.value = value;  
        this.next = next;  
    }  
}
```

Если в такой структуре нужно удалить элемент, то можно перенаправить ссылку на следующий элемент за удаляемым элементом (рис.7.2). Удаляемый элемент со значением 12 будет удален сборщиком мусора (*garbage collector*), который удаляет все объекты из кучи, на которые больше не ссылаются.



Рассмотрим случай вставки элемента в список. Например, нужно вставить элемент 25 между элементами 47 и 12. В этом случае ссылку элемента 47 нужно направить на элемент 25, а ссылку элемента 25 на 12 (рис.8.3).



Посмотрим несколько реализаций такой структуры.

```

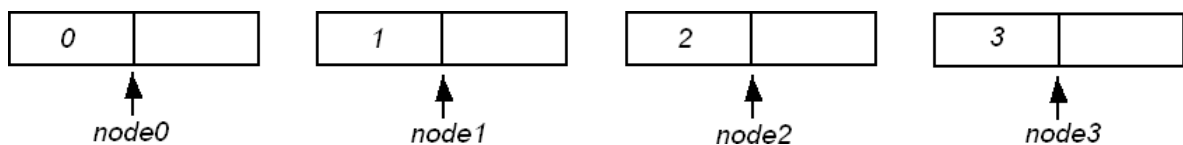
public class SingleLinkedList1 {
public static void main (String[] args) {
    Node node0 = new Node(0,null);
    Node node1 = new Node(1,null);
    Node node2 = new Node(2,null);
    Node node3 = new Node(3,null);

    node0.next = node1;
    node1.next = node2;
    node2.next = node3;

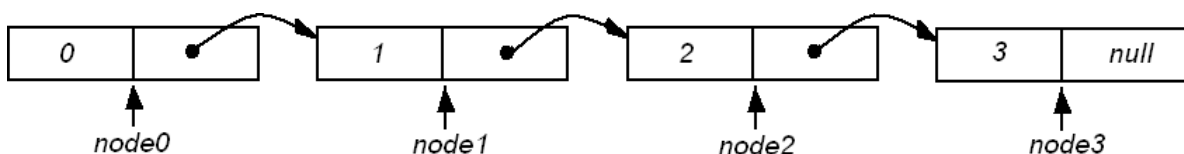
    Node ref = node0;
    while (ref != null) {
        System.out.print("" + ref.value);
        ref = ref.next;
    }
}
}

```

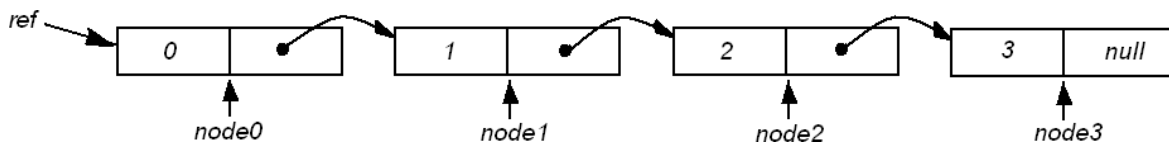
В программе сначала создаются элементы списка (экземпляры класса Node)



Потом задаются ссылки node0 на node1, node1 на node2, node2 на node3.



Далее создается ссылка ref указывающая на node0.



В цикле, до тех пор пока `ref` не равен `null`, распечатывается значение `value` того экземпляра класса, на который указывает ссылка `ref`. В цикле же выполняется оператор `ref = ref.next`. Таким образом, мы переносим ссылку `ref` на `ref.next`, т.е. на новый элемент. Так в цикле ссылка `ref` сначала будет ссылаться на `node0`, в следующей итерации на `node1`, затем на `node2`, и на `node3`. Так как `node3` не ссылается на какой-либо элемент, поле `next` этого элемента равно `null`. Как только `ref` станет равен `null` цикл завершится. С помощью данного цикла был осуществлен проход по списку. Такой список называется односвязным. Это динамическая структура данных. На это указывает и цикл прохода по списку. В нем нигде не указан его размер. Переменная `ref` в программировании называется курсор или итератор.

Представим себе, что теперь нам нужно создать список, состоящий из 1000 элементов. Предыдущий код будет тяжело применить для такой задачи. Давайте изменим его. Первым шагом к изменению будет запись вида:

```
Node node3 = new Node(3,null);
Node node2 = new Node(2,node3);
Node node1 = new Node(1,node2);
Node node0 = new Node(0,node1);
```

В этом примере ссылка на соседний элемент указывается сразу при создании текущего элемента. Так, например, при создании элемента `node2` в конструктор передается целочисленное значение 2 для поля `value` и ссылка на `node3` для поля `next`. Код стал короче, но проблему с созданием большого количества элементов он все равно не решает.

Еще один интересный вариант:

```
Node node0 = new Node(0, new Node(1, new Node(2, new Node(3,null))));
```

Тут уже нет явных ссылок. Явно создана только ссылка `node0`, а остальные создаются анонимно. При выполнении такого сложного конструктора JVM будет поочередно создавать элементы. Сначала будет создан элемент со значением 3, потом со значением 2, со значением 1 и в конце `node0`.

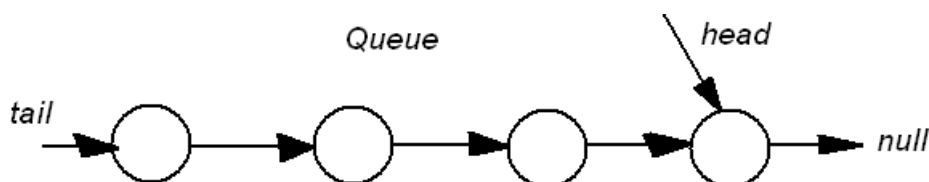
Еще одна попытка – фабричный метод:

```
Node node0 = node(0, node(1, node(2, node(3,null))));
static Node node(int value, Node next){
    return new Node(value, next);}
```


Для параметрического создания больших связанных списков, лучше всего подойдет цикл:

```
int n = 1000;
Node tail = null;
for (int k = 0; k < n; k++) {
    tail = new Node(k, tail);
}
```

Теперь мы можем создавать в цикле сколь угодно много элементов списка, задав их количество в переменной *n*. Хвостовой элемент в примере назван *tail*. Такая структура называется очередь (Queue). Она полностью имитирует очередь людей. Последний в очереди всегда знает, кто стоит перед ним, но не знает кто за ним. Обычно последний (хвостовой) элемент очереди принято называть *tail*, а первый – *head*.



Обратите внимание на присваивание в цикле: `tail = new Node(k, tail)`. Инструкция выполняется слева направо: сначала вызов конструктора, а потом присваивание. В первой итерации цикла будет вызван конструктор `Node` с аргументами `k=0` и `tail=null`. Во второй итерации `k=1`, а в `tail` будет уже сохранена ссылка на предыдущий элемент со значением 0. В оставшихся итерациях данный алгоритм повторится, при создании элемента в конструктор будет передано целое число равное текущему значению параметра `k` и ссылка на предыдущий элемент.

Рассмотрим задачу, в которой создадим односвязный список и распечатаем его элементы двумя способами: итеративно и рекурсивно.

Задача 8.6 *Создайте односвязный список. Разработайте два метода обхода элементов списка: рекурсивный и итеративный.*

Объяснение: В программе создается односвязный список длиной 10. И два метода: `toStringIter()` и `toStringRec`, которым в качестве аргумента передается ссылка на хвостовой элемент `tail`. Первый метод реализует проход по элементам списка итеративно, второй – рекурсивно.

Рассмотрим, как работает итеративный метод `toStringIter()`. В метод передается аргумент – ссылка на хвост списка `tail`, которая копируется в переменную `ref`. В методе объявлена строковая переменная аккумулятор `result`, первоначально представляющая собой пустую строку. В

дальнейшем в нее будут накапливаться элементы списка (по одному в каждой итерации). Переменная result – это еще и возвращаемое методом значение. В цикле while до тех пор, пока ссылка ref не равна null, выбирается каждый следующий элемент (с помощью инструкции ref=ref.next) и конкатенируется с переменной result. Обратите внимание, что цикл может оказаться бесконечным, если список будет иметь круговые ссылки (например, хвостовой элемент будет иметь ссылку на начальный элемент списка). В этом случае ref никогда не будет равна null.

Метод toStringRec() выполняет ту же задачу, что и метод toStringIter() только решает рекурсивно, вызывая для этого сам себя. Условием выхода из рекурсии будет условие равенства ссылки ref значению null. Если ссылка ref не равна null, то значение value, на которое она указывает, добавляется к строке и вызывается метод toStringRec() для оставшейся части списка.

```
//Класс, описывающий элемент списка
public class Node {
    public int value;
    public Node next;

    Node(int value, Node next) {
        this.value = value;
        this.next = next;
    }
}

public class SingleLinkedListTest {
    public static void main(String[] args) {
        Node tail = null;
        for (int k = 0; k < 10; k++) {
            tail = new Node(k, tail);
        }

        System.out.println(toStringIter(tail));
        System.out.println(toStringRec(tail));
    }

    public static String toStringIter(Node ref) {
        String result = "";
        while (ref != null) {
            result += "(" + ref.value + ")->";
            ref = ref.next;
        }
        result += "null";
        return result;
    }

    public static String toStringRec(Node ref) {
```

```

        if (ref == null) {
            return "null";
        } else {
            return "(" + ref.value + ")->" + toStringRec(ref.next);
        }
    }
}

```

Результат:

```

>> (9)->(8)->(7)->(6)->(5)->(4)->(3)->(2)->(1)->(0)->null
>> (9)->(8)->(7)->(6)->(5)->(4)->(3)->(2)->(1)->(0)->null

```

Рассмотрим пример, в котором односвязные списки генерируются рекурсивно и итеративно.

Задача 8.7 Сгенерируйте односвязные списки, используя для этого рекурсивный и итерационный методы. Предусмотрите возможность генерирования списков в прямом и обратном порядках.

//Класс, описывающий элемент списка

```

public class Node {
    public int value;
    public Node next;

    Node(int value, Node next) {
        this.value = value;
        this.next = next;
    }
}

public class SingleLinkedListTest {
    public static void main(String[] args) {
        Node tailA = generateIter0(9);
        Node tailB = generateIter1(9);
        Node tail0 = generateRec0(9);
        Node tail1 = new Node(9, null); generateRec1(tail1);

        //Вывод списков на экран
        System.out.println(toStringRec(tailA));
        System.out.println(toStringRec(tailB));
        System.out.println(toStringIter(tail0));
        System.out.println(toStringIter(tail1));
    }

    //Метод генерирует список итеративно с головы
    public static Node generateIter0(int max) {

```

```

        Node result = null;
        for (int k = 0; k <= max; k++) {
            result = new Node(k, result);
        }
        return result;
    }

    //Метод генерирует список итеративно с хвоста
    public static Node generateIter1(int max) {
        Node tail = new Node(max, null);
        Node head = tail;
        for (int k = max; k > 0; k--) {
            head.next = new Node(k - 1, null);
            head = head.next;
        }
        return tail;
    }

    //Метод генерирует список рекурсивно с головы
    public static Node generateRec0(int value) {
        if (value >= 0) {
            return new Node(value, generateRec0(value - 1));
        } else {
            return null;
        }
    }

    //Метод генерирует список рекурсивно с хвоста
    public static void generateRec1(Node node) {
        if (node.value > 0) {
            node.next = new Node(node.value - 1, null);
            generateRec1(node.next);
        }
    }

    public static String toStringIter(Node ref) {
        String result = "";
        while (ref != null) {
            result += "(" + ref.value + ")->";
            ref = ref.next;
        }
        result += "null";
        return result;
    }

    public static String toStringRec(Node ref) {
        if (ref == null) {
            return "null";
        } else {
            return "(" + ref.value + ")->" + toStringRec(ref.next);
        }
    }
}

```

Упражнения к главе 8

1. Используя динамическую структуру «стек» для хранения символов, определите, является ли заданная фраза палиндромом, то есть такой, которая одинаково читается в обоих направлениях (слева направо и справа налево).
2. Используя динамическую структуру список, сохраните ряд чисел. Удалить элементы, которые повторяются.
3. Сформируйте динамическую структуру очередь, элементами которой являются цифры. Извлекая элементы из очереди, напечатайте их двоичные эквиваленты.
4. Постройте динамический список из элементов целого типа с помощью генератора случайных чисел. Найдите среднее арифметическое значение элементов списка.
5. Задайте два динамических списка. Проверьте их на равенство.
6. Используя динамическую структуру список, подсчитайте количество положительных и отрицательных чисел в списке.
7. Используя динамическую структуру «стек», зашифруйте содержание текстового файла.
8. Сформируйте очередь из элементов целого типа. Четные элементы возведите в квадрат. Распечатайте исходную и результирующие очереди.
9. Используя динамическую структуру список, проверьте является ли он упорядоченным набором чисел.
10. Используя динамическую структуру очередь, перевести введенную последовательность цифр в слово, состоящее из кодов Unicode.

Навчальне електронне видання

КОВАЛЕНКО ЛЮДМИЛА БОРИСІВНА
КУЗНІЧЕНКО СВІТЛАНА ДМИТРІВНА

АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ

Конспект лекцій

Видавець і виготовлювач

Одеський державний екологічний університет

вул. Львівська, 15, м. Одеса, 65016

тел./факс: (0482) 32-67-35

Е-mail: info@odeku.edu.ua

Свідоцтво суб'єкта видавничої справи

ДК № 5242 від 08.11.2016