МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

КУЗНІЧЕНКО С.В.

КРОС-ПЛАТФОРМНЕ ПРОГРАМУВАННЯ

Конспект лекцій

Одеса
Одеський державний екологічний університет
2016

УДК 681.3
К89

Рекомендовано методичною радою Одеського державного екологічного університету Міністерства освіти і науки України як конспект лекцій (протокол №10 від 04.07. 2016 р.)

Кузніченко С.Д.
Крос-платформне програмування: конспект лекцій. Одеса, Одеський державний екологічний університет, 2016. 100 с.

В конспекті лекцій розглянуто фундаментальні принципи компонентної та розподіленої організації програм, а також прийоми практичного створення компонентних розподілених програмних продуктів на широко розповсюдженій мові крос-платформного програмування Java.

Рекомендовано для студентів галузі знань 12 "Інформаційні технології" першого (бакалаврського) рівня.

ISBN 978-966-186-092-5

ЗМІСТ

ВСТУП

В конспекті лекцій з дисципліни «Крос-платформне програмування» викладається теоретичний матеріал щодо засобів крос-платформного програмування та створення з їх використанням відповідних програмних систем на мові програмування Java.

Розглянуті найважливіші аспекти застосування бібліотек класів мови Java, включаючи файли, колекції, мережеві і багатопотокові застосування.

Конспект лекцій «Крос-платформне програмування» складається з 4 розділів. В кінці кожного розділу надаються контрольні питання за матеріалом глави і завдання для виконання.

Крос-платформне програмне забезпечення — програмне забезпечення, що працює більш ніж на одній апаратній платформі і операційній системі (ОС).

Крос-платформне програмування – технологія створення і інтеграції в єдину систему *компонентів*, які розроблені на різних платформах.

Поняття кросплатформності може використовуватися на різних рівнях абстракції інформаційних систем:

1) На рівні мови програмування

Крос-платформними можна назвати більшість сучасних мов програмування високого рівня. Наприклад, C, C++ і Object Pascal — крос-платформні мови на рівні компіляції, тобто для цих мов є компілятори під різні платформи. Java і C# — крос-платформні мови на рівні виконання, тобто їх виконувані файли можна запускати на різних платформах без попередньої перекомпіляції. Мови скриптів – PHP, ActionScript, Perl, Python, Tcl і Ruby — кросплатформні мови, що інтерпретуються, їх інтерпретатори існують для багатьох платформ.

2) На рівні прикладних програм

Багато прикладних програм також є крос-платформними. Особливо ця якість виражена в програмах, спочатку розроблених для UNIX-подібних операційних систем. Важливою умовою їх переносності на інші платформи є сумісність платформ з рекомендаціями POSIX, а також існування компілятора для платформи, на яку здійснюється перенесення.

3) На рівні операційної системи

Сучасні операційні системи також часто є крос-платформними. Наприклад, операційні системи з відкритим кодом, такі як NETBSD, Linux,

FREEBSD, AROS можуть працювати на декількох різних платформах, найчастіше це x86, m68k, POWERPC, Alpha, AMD64, SPARC. Microsoft Windows може працювати як на платформі Intel x86, так і на Intel Itanium.

В даному конспекті лекцій зупинимося більше докладно на поняттях кросплатформності на рівні мови програмування, на прикладі сучасної мови програмування Java.

1 БАГАТОПОТОКОВЕ ПРОГРАМУВАННЯ

Мова Java є однією з небагатьох мов програмування, які містять засоби підтримки потоків. Потоки можна застосувати в будь-якій програмі при необхідності паралельного виконання декількох завдань. Так, наприклад, до більшості сучасних розподілених додатків (Rich Client) і Web-додатків (Thin Client) висуваються вимоги одночасної підтримки багатьох користувачів, кожному з яких виділяється окремий потік, а також розділення і паралельної обробки інформаційних ресурсів. Потоки – це засіб, який допомагає організувати одночасне виконання декількох завдань, кожне в незалежному потоці. Потоки являють собою класи, кожен з яких запускається і функціонує самостійно, автономно (або відносно автономно) від головного потоку виконання програми.

Використання потоків дає можливість писати дуже ефективні програми, які максимально використовують CPU, тому що час його простою можна звести до мінімуму. Це особливо важливо для інтерактивного мережного середовища, в якій працює Java, бо час простою є загальним.

Як відомо, одноядерний процесор обробляє інструкції наступним чином: в певний момент часу до процесора надходить чергова інструкція, яка, залежно від свого пріоритету та інших параметрів, що залежать від архітектури процесора, стає в чергу. Процесор послідовно виконує кожну інструкцію (тут простий опис, в реальності все залежить від архітектури – процесор може виконувати одну інструкцію, потім по перериванню переключитися на іншу, виконати її і повернутися до колишньої). Це говорить про те, що одноядерний процесор обробляє весь потік команд, що надходять, ПОСЛІДОВНО, навіть якщо програма написана як багатопотокова. Тому слід чітко розуміти, що паралельного виконання потоків на одноядерному процесорі не буде. Хоча за рахунок більш оптимального використання часу простою процесора в багатопотоковому застосуванні в порівнянні з однопотоковим, швидкість виконання програми, безумовно, може стати вище.

Інтерес до багатопоточного програмування виріс після 2005 року, коли стало зрозуміло, що підвищити продуктивність комп'ютерів за рахунок підвищення тактової частоти процесора очікувати не доводиться. З'явилися багатоядерні процесори (кілька окремих процесорів на одному кристалі). Потоки, що надходять на багатоядерний процесор інструкцій, обробляються

ПАРАЛЕЛЬНО ядрами цього процесора, тобто, черга розбивається на кількість черг рівних кількості ядер (взагалі ж це залежить від певного процесора).

1.1 Реалізація потоків в Java

Реалізація потоків в програмах може виконуватися двома способами:

- розширенням класу **Thread**;
- реалізацією інтерфейсу **Runnable**.

При першому способі клас стає потоковим, якщо він створений як розширення класу Thread , який визначений в пакеті java.lang, наприклад:

public class GreatRace extends Thread

При цьому стають доступними всі методи потоків.

Зазвичай, коли необхідно, щоб даний клас був розширенням деякого іншого класу і в ньому необхідно реалізувати потоки, попередній підхід не можна використовувати, оскільки в Java немає множинного спадкоємства. Для вирішення цієї проблеми для даного класу потрібно реалізувати інтерфейс Runnable, наприклад:

public class GreatRace extends Applet implements Runnable

Інтерфейс Runnable має тільки один метод **public void run()**. Насправді клас Thread також реалізує цей інтерфейс, проте стандартна реалізація run() у класі Thread не виконує ніяких операцій. Необхідно або розширити клас Thread, щоб включити в нього новий метод run(), або створити об'єкт Runnnable і передати його конструктору потока. Коли створюється клас, який реалізує інтерфейс Runnable, цей клас повинен перевизначити метод run(). Саме цей метод виконує фактичну роботу, покладену на конкретний потік.

Запуск потоку виконує метод **public void start() throws IllegalThreadStateException** (виняток кидається, якщо робиться спроба запуску вже запущеного потоку). Виклик методу start() – це єдиний спосіб в Java породити незалежну активність (при прямому виклику методу run() потік не запуститься, виконається тільки тіло самого методу).

Для зупинки потоку рекомендується привласнити потоку значення null, наприклад:

```
Thread myThread;
…
myThread.start(); // Запуск потоку
…
myThread = null; // Зупинка або завершення потоку
```

Розглянемо декілька прикладів програм, що виводять на консоль із заданою періодичністю літери «A» і «B» в паралельно працюючих потоках.

Варіант 1. Створено два класи, один – розширюється від Thread другий реалізує інтерфейс Runnable.

```
public class Thread1 extends Thread {
public void run() {
for (int i = 0; i < 10; i++) {
System.out.println("A");
try {
Thread.sleep(130);
} catch (InterruptedException e) {
System.err.print(e);
}
}
}
}

public class Thread2 implements Runnable {
public void run() {
for (int  i = 0; i < 10; i++) {
System.out.println("  B");
try {
Thread.sleep(80);
} catch (InterruptedException e) {
System.err.println(e);
}
}
}
}

public class Example {
public static void main(String[] args) {
Thread1 thread1  = new Thread1();
Thread thread2 = new Thread(new Thread2());
thread1.start();
thread2.start();
}
}
```

Варіант 2. Створений клас, що реалізує інтерфейс `Runnable` з конструктором, який отримує в якості аргументів текст для друку і

періодичність його друку у мілісекундах. В методі main() створюються два екземпляри даного класа і запускаються їх методи `run()` в двох параллельно працюючих потоках виконання.

```
public class ExampleRunnable implements Runnable{
private String msg;
private long sleepMillis;

public ExampleRunnable(String msg, long sleepMillis) {
     this.msg = msg;
     this.sleepMillis = sleepMillis;
    }
public void run(){
  for (int i=0; i<10; i++) {
  try{
      Thread.sleep(sleepMillis);
    } catch (InterruptedException e) {
       throw new RuntimeException(e);
    }
    System.out.println(msg);
}
}}

 public class Example {
 public static void main(String[] args)
                        throws InterruptedException{
 Thread thread1 = new Thread (new ExampleRunnable("  B",
80));
 Thread thread2 = new Thread (new ExampleRunnable("A",
130));
 thread1.start();
 thread2.start();

}}
```

1.1.1 Методи sleep() і yield()

Якщо потрібно, щоб перед продовженням роботи потік чекав визначений час, можна використовувати метод **public static void sleep(long millis) throws InterruptedException** або **public static void sleep(long millis , int nanos) throws InterruptedException**. Метод sleep() дуже часто використовується в циклах, керуючих анімацією.

Якщо в програмі є потік, який захоплює процесор для великої кількості обчислень, може з'явитися необхідність змусивати його час від часу "відпускати" процесор, даючи можливість виконуватися іншим потокам. Це досягається за допомогою методу **public static void yield()**.

Виклик методу yield() для виконуваного потоку повинен призводити до припинення потоку на деякий квант часу, для того щоб інші потоки могли виконувати свої дії. Однак якщо потрібна надійна зупинка потоку, то слід використовувати його вкрай обережно або взагалі застосувати інший спосіб.

```
public class YieldRunner {
public static void main(String[] args) {
new Thread() {
         public void run() {
         System.out.println("старт потоку 1");
         Thread.yield();
         System.out.println("завершення 1");
        }
     }.start();
new Thread() {
         public void run() {
         System.out.println("старт потоку 2");
         System.out.println("завершення 2");
         }
    }.start();
}}
```

В результаті може бути виведено:

```
старт потоку 1
старт потоку 2
завершення 2
завершення 1
```

Активізація методу yield() в коді методу run() першого об'єкта потоку призведе до того, що перший потік буде зупинений на деякий квант часу, що дасть можливість іншому потоку запуститися і виконати свій код.

1.1.2 Метод join()

Метод **public final void join() throws InterruptedException** блокує роботу потока, в якому він викликаний, поки не буде закінчено виконання потоку, що викликає цей методу. Розглянемо приклад:

```
public class Example {
     public static void main(String[] args)
                          throws InterruptedException{
   Thread thread1 = new Thread (new ExampleRunnable("  B", 80));
   Thread thread2 = new Thread (new ExampleRunnable("A", 130));
```

```
        thread1.start();
        thread1.join();
        thread2.start();
}}
```

У прикладі спочатку будуть виведені 10 літер "B", а тільки потім 10 літер "A". На самому початку роботи методу `main`() він запустить новий потік `thread1`, а після виклику методу `thread1.join()` причепиться до нього і буде чекати його завершення. Як тільки потік `thread1` завершить свою роботу метод `main()` почне виконуватися далі, а саме, запустить потік `thread2` на виконання.

Такий механізм часто потрібен тоді, коли в програмі паралельно виконується кілька потоків, що вирішують окремі підзадачі алгоритму і методу main() потрібно дочекатися результатів їх виконання, щоб виконати остаточні розрахунки і завершити алгоритм.

Як ви можете бачити метод `join`() викликається для екземпляра класу Thread. Щоб отримати посилання на той потік, в якому ми знаходимося, слід викликати статичний метод `currentThread`() класу `Thread`.

Наступний код призведе до того, що програма повисне, оскільки метод main() буде очікувати завершення (смерті) самого себе. Таке явище в багатопотоковому програмуванні називається *взаємне блокування* або *dead lock*.

```
    public static void main(String[] args)
                          throws InterruptedException{
     Thread thread = Thread.currentThread();
     thread.join();
     }
```

Така ж ситуація може виникнути якщо один потік прив'язується до іншого і очікує його смерті, в той час як цей другий потік прив'язався до першого і теж чекає його смерті. У підсумку маємо dead lock. Два потоки чекають завершення один одного.

Dead lock можна вирішити шляхом використання **public final syncronized void join(long millis) throws InterruptedException** і **public final syncronized void join(long millis, int nanos) throws InterruptedException**. Ці методи чекають протягом millis мілісекунд або nanos наносекунд. Якби в прикладі було записано thread.join (1000), то через 1000 мс потік методу main() відпустило б і він продовжив своє виконання.

1.2 Синхронізація потоків

Основна відмінність потоків від процесів полягає в тому, що потоки не захищені один від одного засобами операційної системи. Тому будь-який з потоків може отримати доступ і навіть внести зміни в дані, які інший потік вважає своїми. Вирішення цієї проблеми полягає в синхронізації потоків.

Модель синхронізації потоків – монітор – це механізм управління зв'язком між потоками придуманий Хором (Hoare, CAR). Монітор можна представити як маленький блок, який містить тільки один потік. Як тільки потік входить в монітор, всі інші потоки повинні чекати, поки даний не вийде з монітора. Таким чином, монітор можна використовувати для захисту спільно поділюваних ресурсів від управління декількома потоками одночасно. Тобто синхронізація потоків полягає в гарантії надання доступу до даних тільки для одного потоку в кожен момент часу.

Синхронізувати код можна двома способами. Обидва використовують ключове слово synchronized. Перший спосіб – створити синхронізований блок. Наприклад, так

synchronized (*вираз*) {
оператори для синхронізації
}

Взятий в дужки *вираз* повинен вказувати на дані, що блокуються, – зазвичай він є посиланням на об'єкт. Об'єкт не повинен бути рівний null.

Другий спосіб – записати слово synchronized в сигнатурі методу (у розділі модифікаторів). Це потрібно, якщо виявляється, що критична ділянка поширюється на весь метод, а ресурсом, що розділяється, є весь об'єкт в цілому, наприклад:

```
synchronized void myMethod()
{
// тіло методу
}
```

Більш гнучкий і ефективний спосіб координації виконання потоків забезпечують методи **wait()** і **notify()** класу **Object.**

Метод **wait()** переводить потік в стан очікування виконання певної умови і викликається за допомогою однієї з наступних форм:

public final void wait() throws InterruptedException, IllegalMonitorStateException – зупинення виконання поточного потоку до отримання повідомлення;

public final void wait(long timeout) throws InterruptedException, IllegalMonitorStateException, IllegalArgumentException – зупинення виконання поточного потоку до отримання повідомлення або до закінчення заданого інтервалу часу timeout (в мілісекундах);

public final void wait(long timeout, int nanos) throws InterruptedException, IllegalMonitorStateException, IllegalArgumentException – зупинення виконання поточного потоку до отримання повідомлення або до закінчення заданого інтервалу часу timeout (в мілісекундах) і, додатково, в наносекундах (значення в діапазоні 0-999999).

Метод **public final void notify()** переводить в активний стан один з потоків, встановлених в стан очікування за допомогою методу wait(). Критерій вибору потоку є довільним і залежить від конкретної операційної системи і реалізації віртуальної машини Java. Якщо необхідно перевести всі потоки, що очікують, в активний стан, можна скористатися **методом public final void notifyAll()**.

Методи можуть викликатися тільки потоком, який є власником монітора даного об'єкта. Якщо до виклику методів wait() або notify() не виконати захоплення монітора даного об'єкта, генерується виключення **IllegalMonitorStateException**. Потік стає власником даного об'єкта одним з трьох способів:

- викликом методу, оголошеного як **synchronized**, який здійснює доступ до необхідного екземпляра об'єкта класу **Object**;
- виконанням тіла оператора **synchronized**, призначеного для синхронізації доступу до необхідного екземпляра об'єкта класу Object;
- виконанням, для об'єктів типу Class, синхронізованого статичного методу цього класу.

Розглянемо на простих прикладах правила написання синхронізованих блоків і методів.

Нижній фрагмент коду коректний, оскільки виклик методу wait() відбувається в блоці, синхронізованому по об'єкту obj0. У цьому же місці можна викликати і метод wait() для об'єкта obj1. В Java можна створювати

будь-яку кількість вкладених синхронізованих блоків. При цьому потік запам'ятовує, в яку кількість синхронізованих блоків він увійшов.

```
public static void main(String[] args)
                          throws InterruptedException{
          Object obj0 = new Object();
          Object obj1 = new Object();
             synchronized (obj0) {
               synchronized (obj1) {
                    obj0.wait();
             }
          }
       }
```

Наступний код абсолютно коректний. У синхронізованому нестатичному методі f() можна викликати метод notify(). Пам'ятайте, що статичний метод не має посилання this!

```
public class Example{
public static void main(String[] args)
                          throws InterruptedException{
      new Example().f();
    }

public synchronized void f() {
      this.notify();
    }
}
```

Попередній код повністю еквівалентний наступному:

```
public class Example{
public static void main(String[] args)
                          throws InterruptedException{
      new Example().f();
    }

public void f() {
      synchronized (this) {
      this.notify();
    }
  }
}
```

Але як бути, якщо існує необхідність викликати методи wait(), notify() або notifyAll() в статичному методі. Як відомо у статичного метода немає

посилання this. У статичному методі wait() і notify() можна викликати для об'єкта класу, як це показано нижче:

```
public class Example{
public static void main(String[] args)
                          throws InterruptedException{
      f();
    }

public static synchronized void f() {
      Class clazz = Example.class;
      clazz.notify();
    }
  }
```

Такий запис цілком коректний. Клас є посилальним типом даних, тобто об'єктом для якого може бути викликаний метод notify().

Можна видозмінити наведений вище код, використовуючи синхронізований блок:

```
public class Example{
public static void main(String[] args)
                          throws InterruptedException{
      f();
    }

public static void f() {
      Class clazz = Example.class;
      synchronized (clazz) {
      clazz.notify();
    }
  }
}
```

Розглянемо приклад. Мається клас Blocked Method Caller, який реалізує інтерфейс Runnable. У конструктор даного класу передається посилання на об'єкт і ціле число k. У методі run() для даного посилання викликається метод f(), в який і передається ціле число k.

Окремо створюється клас BlockedSetExample, що має метод f(). Ідея полягає в тому, щоб створювати багато потоків (у прикладі їх 5), кожен з яких може викликати синхронізований метод f().

ПРИКЛАД 3

```
public class BlockedMethodCaller implements Runnable {
private final BlockedSetExample ref;
private final int k;
```

```
public BlockedMethodCaller (BlockedSetExample ref, int k) {
this.ref = ref;
this.k = k;
}
@Override
public void run() {
   try {
      ref.f(k);
} catch (InterruptedException e) {
     e.printStackTrace();}
}}
```

```
public class BlockedSetExample {
public static void main (String[] args) throws
InterruptedException {
  BlockedSetExample ref = new BlockedSetExample();
  for (int k = 0; k < 5; k++) {
   new Thread(new BlockedMethodCaller(ref, k)).start();
}}
```

```
public synchronized void f(int x) throws InterruptedException {
   System.out.println("+"+x);
   Thread.sleep(1000);
   System.out.println("-"+x);
}}
```

У програмі створюється 5 потоків, які викликають синхронізований метод f() (рис. 1.1). У самому методі f() потік засинає на 1 сек. Кожен потік має свій номер – ціле число x.

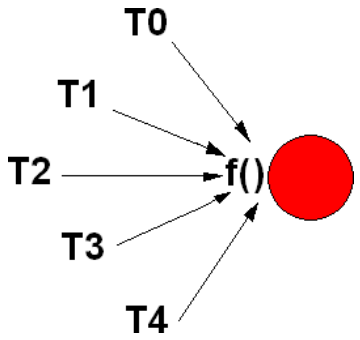

Рисунок 1.1 – Демонстрація роботи програмного кода для прикладу 3

При вході потоку в метод f () на екран виводиться його номер зі знаком "+", після того як потік прокинеться, на екран виводиться його номер, але вже зі знаком "–". Результат роботи програми наводиться нижче:

```
+0
-0
+4
-4
+3
-3
+1
-1
+2
-2
```

Як видно з результату роботи програми, потоки входять в метод f() по черзі. Поки в методі спить один потік, ніякий інший потік в метод зайти не може. Як тільки потік, що знаходиться в методі, прокидається і покидає його, в метод заходить наступний потік. Таким чином, ніякі два і більше потоків не можуть знаходитися і одночасно працювати в синхронізованій по одній змінній секції. Це властивість називається *взаємне виключення* (mutual exclusion) або *мьютекс*.

У прикладі потоки синхронізуються по одному посиланню ref. У метод f() воне передається неявно як посилання this. Спробуємо створити для кожного потоку окреме посилання:

ПРИКЛАД 4

```
public class BlockedSetExample {
public static void main (String[] args) throws
InterruptedException {
   for (int k = 0; k < 5; k++) {
   new Thread(new BlockedMethodCaller(new BlockedSetExample(),
k)).start();
}}

public synchronized void f(int x) throws InterruptedException {
   System.out.println("+"+x);
   Thread.sleep(1000);
   System.out.println("-"+x);
}}
```

В цьому випадку ми не спостерігаємо синхронізацію потоків, незважаючи на те, що в заголовку методу f() записано слово synchronized. Результат виконання програми має наступний вигляд:

```
+2
+0
+1
+3
+4
-0
-2
```

```
-4
-3
-1
```

Це пов'язано з тим, що тепер ми створюємо п'ять різних об'єктів і кожен потік викликає синхронізований метод f() у свого об'єкта (рис.1.2). Так як у кожного об'єкта по одному потоку, то конкурувати нікому. Потоки одночасно можуть зайти кожен у свій метод, поспати 1 сек і прокинутися. Тому на екрані спочатку з'явилися всі плюси, а потім мінуси. Пам'ятайте, що синхронізація здійснюється по об'єктах!

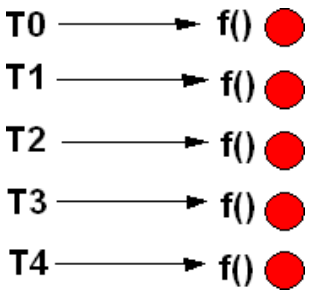

Рисунок 1.2 – Демонстрація роботи прогрпмного коду для приклада 4

Можете поекспериментувати і прибрати з заголовка методу f() слово synchronized. Результат буде наступний:

```
+0
+3
+2
+4
+1
-0
-4
-2
-3
-1
```

Поведінка потоків змінилася. Зверніть увагу, незважаючи на те, що стартували потоки послідовно, починаючи з нульового, добігли до методу f() вони в довільному порядку (і прокинулися теж). Це один із прикладів непередбачуваної поведінки потоків.

Розглянемо ще один приклад:

ПРИКЛАД 5

```
public class WaitMethodCaller implements Runnable {
private final WaitSetExample ref;
private final int k;

public WaitMethodCaller (WaitSetExample ref, int k) {
this.ref = ref;
```

```
this.k = k;
}
@Override
public void run() {
   try {
      ref.f(k);
} catch (InterruptedException e) {
     e.printStackTrace();}
}}

public class WaitSetExample {
public static void main (String[] args) throws
InterruptedException {
  WaitSetExample ref = new WaitSetExample();
  for (int k = 0; k < 5; k++) {
   new Thread(new BlockedMethodCaller(ref, k)).start();
}}

public synchronized void f(int x) throws InterruptedException {
   System.out.println("+"+x);
   this.wait();
   System.out.println("-"+x);
}}
     Результат:
     +0
     +2
     +3
     +4
     +1
```

У синхронізованому методі f() потоки тепер не сплять 1 сек., а переводяться в режим очікування методом wait(). Коли потік в режимі очікування, то розбудити його можуть тільки методи notify() і notifyAll(), яких в програмі немає. Як ви можете бачити по результату виконання програми, всі п'ять потоків змогли зайти в синхронізований по одному об'єкту метод і кожен з них окремо повис. Як же можна пояснити такий дивний результат, адже вище вже говорилося про те, що одночасно не більше одного потоку можуть перебувати і працювати в синхронізованої секції? Правильно, працювати, але не викликати метод wait()! Єдиний спосіб звільнити блокування синхронізованої секції – викликати для потоку метод wait(). У цьому випадку потоки переходять в окрему множину, що зветься wait-set.

З будь-яким об'єктом в Java пов'язано трьох незримих множини. Схематично це можна представити наступним чином (рис.1.3):

– ***Блокування.*** Об'єкт може бути або блокований, або не заблокований (у випадку, коли в synchronized по цьому об'єкту хтось увійшов, і ніхто інший вже туди не може увійти).

– ***Множина wait-set.*** Потоки всередині цієї множини пасивні, вони сплять і чекають, коли їх розбудять.

– ***Множина blocked-set.*** Потоки в цій множині ніби активні, але не працюють, їм щось заважає працювати.

Змоделюємо ситуацію з блокуванням в **прикладі 3**. Коли в синхронізований метод увійшов перший потік (№ 0) і встановив блокування, ніякий інший потік туди зайти вже не може, і вони все переміщуються в множину blocked-set (рис.1.3 a). У цій множині потоки активні, але їх тимчасово призупинила ОС, до тих пір, поки не звільниться блокування. Як тільки потік №0 завершить роботу і звільнить блокування, ОС *випадковим чином* запустить у synchronized метод наступний потік з множини blocked-set (рис. 1.3 b).

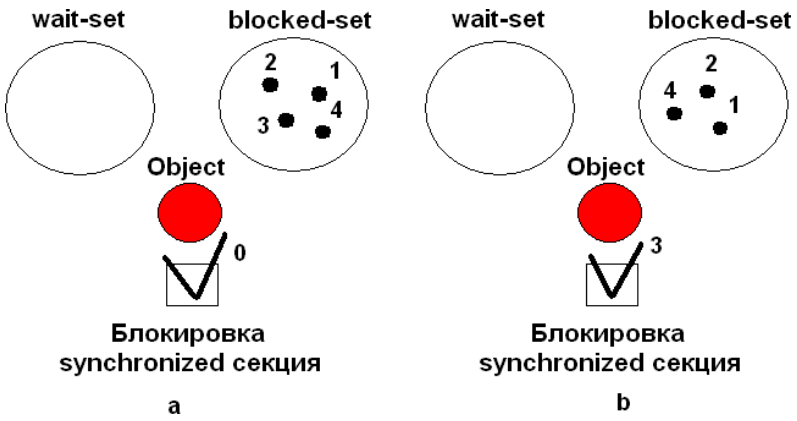

Рисунок 1.3 – Демонстрація роботи програми для прикладу 3

Схематично зобразимо дії, що відбуваються в **прикладі 5**. Потік захоплює синхронізований метод і самостійно викликає метод wait() (рис. 1.4 a). При цьому блокування методу знімається, а сам потік поміщається в множину wait-set. Потім наступний потік ставить блокування і захоплює метод (рис. 1.4 b). Процес повторюється до тих пір, поки всі потоки не опиняться у множині wait-set.

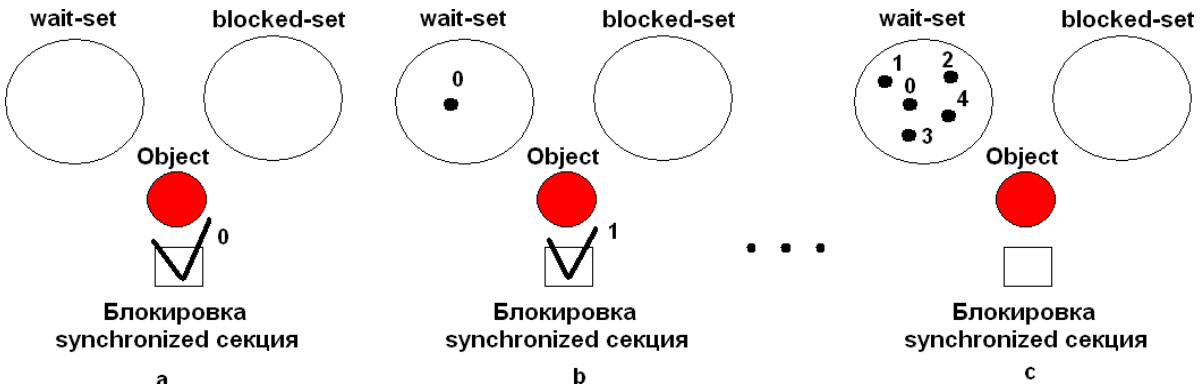

Рисунок 1.4 – Демонстрація роботи програми для прикладу 5

1.2.1 Приклад реалізації простого обмеженого буфера

Розглянемо класичний приклад на взаємодію потоків – обмежений буфер (Bounded buffer).

Обмежений буфер – конкретне уявлення абстрактної ідеї послідовності порцій (рис.1.5). Послідовність (або черга) доступна для багатьох потоків, що паралельно виконуються. Перша їхня група (виробники) модифікують чергу, додаючи до неї послідовно нові порції даних. Друга група (споживачі) модифікують її, забираючи наявні дані. Черга може бути реалізована масивом або списком. Такий патерн називається Виробник-Споживач (Producer-Consumer).

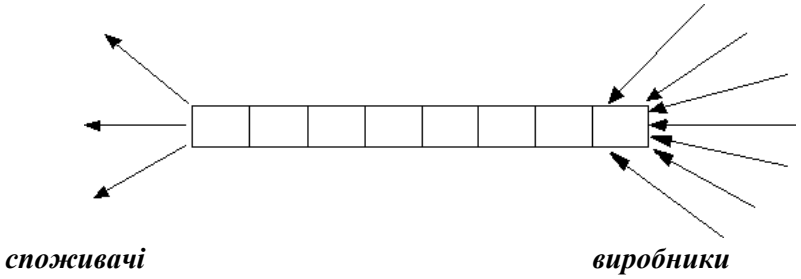

Рисунок 1.5 – Обмежений буфер

Особливістю патерну є те, що кількість потоків-споживачів і потоків-виробників може бути різною. Довжина черги може рости динамічно і фактично є буфером між потоками, що створюють якісь дані і потоками, що їх споживають.

Спробуємо створити подібну чергу. Вона повинна бути потокозахищеною, тобто стійкої до того, що відразу велика кількість потоків хоче до неї писати і забирати дані. Крім того, якщо черга порожня, то потоки

споживачі повинні чекати до тих пір, поки не з'являться дані. Для цього можна використовувати умовне сподівання (conditional wait). Таким чином, якщо черга порожня, всі потоки-споживачі засинають, як тільки з'являються дані, хтось із них прокидається і забирає їх.

Ще один важливий момент – це довжина черги (вона не може бути нескінченно великою інакше буде переповнений heap). Це можливо в тому випадку, коли виробники працюю швидше споживачів. Отже, краще задати заздалегідь максимально можливі розміри черги. Тоді, якщо виробник хоче додати дані у вже повністю заповнену чергу, він повинен блокуватися. Черга сама будить виробників, коли з'являються вільні комірки і будить споживачів, коли з'являються дані.

Тепер спробуємо реалізувати на Java подібний буфер, але тільки для простоти він буде складатися з однієї комірки[1] (рис.1.6).

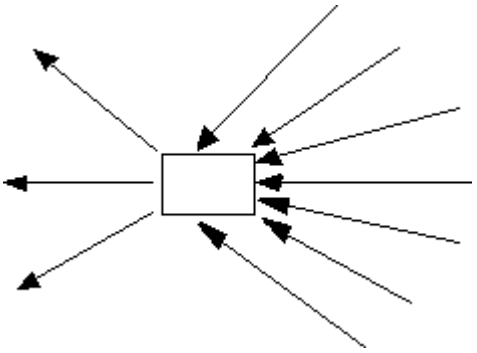

Рисунок 1.6 – Спрощений варіант обмеженого буфера

Спочатку реалізуємо буфер, що складається з однієї комірки, в яку може бути поміщене ціле число. У буфері реалізовано два блокуючих метода: put() і get().

```
public class SingleElementBuffer {
    private Integer elem = null;

    public synchronized void put(Integer newElem)
                                throws InterruptedException {
        while (elem != null) {
            this.wait();
        }
        this.elem = newElem;
        this.notifyAll();
    }
```

[1] З програмним кодом для повноцінного буфера можете ознайомитись за наступним посиланням http://www.java2s.com/Code/Java/Threads/ProducerconsumerinJava2.htm.

```
        public synchronized Integer get()
                                        throws  InterruptedException
{
            while (elem == null) {
                this.wait();
            }
            int result = this.elem;
            this.elem = null;
            this.notifyAll();
            return result;
        }
    }
```

Розглянемо роботу методів post() і get() докладніше.

Якщо виробнику потрібно покласти елемент у буфер, він викликає синхронізований метод put(). Це зручно, тому що буфер складається з однієї комірки, отже, цілком логічно, що в даному методі може перебувати не більше одного потоку. Якщо комірка непорожня (elem! = null), то потік-виробник переводиться в стан очікування (потрапляє в множину wait-set). Трохи вище, розглядаючи приклади кодів, ми вже переконувалися в тому, що єдиний спосіб звільнити блокування синхронізованої секції – викликати для потоку метод wait(). Значить, як тільки блокується поточний потік, зайти в метод put() зможе наступний потік-виробник. Таким чином, усередині синхронізованої секції можуть знаходитися і спати необмежена кількість потоків.

Тепер змоделюємо ситуацію, що з'явився потік-споживач, який викликає метод get(). Для споживача не виконується умова (elem == null), тому він забирає елемент, встановлює значення, рівне null, робить notifyAll() і повертає результат. Коли викликається метод notifyAll(), всі потоки, які були в стані очікування по wait() прокидаються і з множини wait-set переводяться в множину blocked-set. Таким чином, прокинуться всі сплячі потоки-виробники. Але активних потоків усередині синхронізованої секції не може бути більше одного. Тому вони прокинуться всі, але продовжувати роботу зможе тільки один з них. Цей потік-виробник запише в комірку буфера свої дані і викличе метод notifyAll() для потоків-споживачів, що очікують дані. Потім, коли наступний виробник продовжить роботу, комірка, можливо, виявиться знову зайнятої (elem! = null) і він засне по this.wait().

Отже, підсумуємо: при виклику методу notify() один з потоків, обраний випадковим чином, переводиться в множину blocked-set. Якщо викликаний

метод notifyAll(), то всі потоки, що знаходяться в стані wait переходять в blocked-стан. Далі знову ж випадковим чином (алгоритм нечесний – unfair) JVM вибирає з blocked-set потік, який і заходить в синхронізований блок, блокуючи його.

Взагалі ж у blocked-set потік може потрапити двома способами: якщо він очікує в черзі доступу до блокованого іншим потоком sinchronized методу і якщо потік перебував у wait-set і був викликаний метод notify(), який і перевів його в blocked-set. У wait-set можна опинитися одним єдиним способом – потік викликав метод wait().

Створимо виробника. Клас-виробник (producer), виробляє послідовно числа починаючи з startValue (startValue, startValue + 1, startValue + 2, startValue + 3, ...) і поміщає їх в буфер (buffer.put (...)), спить period мілісекунд, повторює (while (true) {...}).

```
     public class Producer implements Runnable {
         private int startValue;
         private final int period;
         private final SingleElementBuffer buffer;

         public    Producer(int    startValue,    int    period,
SingleElementBuffer buffer) {
             this.buffer = buffer;
             this.period = period;
             this.startValue = startValue;
         }

         @Override
         public void run() {
             while (true) {
              try {
               System.out.println(startValue + " produced");
               buffer.put(startValue++);
               Thread.sleep(period);
                 } catch (InterruptedException e) {

System.out.println(Thread.currentThread().getName()      +      "
stopped.");
                     return;
                 }
             }
         }
     }
```

Створимо споживача. Клас-споживач (consumer), з максимальною швидкістю вилучає числа з буфера (buffer.get ()), виводить на консоль, повторює (while (true) {...}).

```
public class Consumer implements Runnable {
    private final SingleElementBuffer buffer;

    public Consumer(SingleElementBuffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        while (true) {
            try {
                int elem = buffer.get();
                System.out.println(elem + " consumed");
            } catch (InterruptedException e) {

System.out.println(Thread.currentThread().getName()      +      "
stopped.");
                return;
            }
        }
    }
```

Протестуємо роботу програми. Система з одним споживачем відразу ж блокується (споживач висить на черзі чекаючи даних):

```
public class ProducerConsumerExample{
 public static void main(String[] args) {
  SingleElementBuffer buffer = new SingleElementBuffer();
    new Thread(new Consumer(buffer)).start();
    }
}
```

>> ... повисла

Система з одним виробником блокується пізніше (встигає виробити 2 числа і повисає намагаючись помістити друге число в чергу вже зайняту перший числом):

```
public class  ProducerConsumerExample{
    public static void main(String[] args) {
        SingleElementBuffer       buffer       =       new
SingleElementBuffer();
        new Thread(new Producer(1, 1000, buffer)).start();
```

```
        }
    }
```

>> 1 produced
... затримка 1 секунда
>> 2 produced
... повисла

Система з одного споживача і одного виробника працює стабільно:

```
    public class ProducerConsumerExample {
        public static void main(String[] args) {
            SingleElementBuffer buffer = new
SingleElementBuffer();
            new Thread(new Producer(1, 1000, buffer)).start();
            new Thread(new Consumer(buffer)).start();
        }
    }
```

>> 1 produced
>> 1 consumed
... затримка 1 секунда
>> 2 produced
>> 2 consumed
... затримка 1 секунда
>> 3 produced
>> 3 consumed
... затримка 1 секунда
>> 4 produced
>> 4 consumed
... і так далі

Система з 3-ма виробниками (з різною швидкістю розміщення елементів у буфер –300, 500 і 700 мілісекунд) і 2-ма споживачами працює з "легкими ривками":

```
    public class ProducerConsumerExample{
        public static void main(String[] args) {
            SingleElementBuffer buffer = new
SingleElementBuffer();
        Thread[] producers = new Thread[]{
        new Thread(new Producer(1, 950, buffer)),
        new Thread(new Producer(100, 1550, buffer)),
        new Thread(new Producer(1000, 2010, buffer)),
            };
            for (Thread producer : producers) {
            producer.start();
            }
            Thread[] consumers = new Thread[]{
```

```
            new Thread(new Consumer(buffer)),
            new Thread(new Consumer(buffer)),
           };
           for (Thread consumer : consumers) {
               consumer.start();
           }
       }
   }
```

```
>>100 produced
>>1000 produced
>>1 produced
>>100 consumed
>>1 consumed
>>1000 consumed
>>2 produced
>>2 consumed
>>101 produced
>>101 consumed
>>3 produced
>>3 consumed
>>1001 produced
>>1001 consumed
... і так далі
```

1.2.2 Ключеве слово volatile

Визначення змінної з ключовим словом volatile означає, що значення змінної буде змінюватися різними потоками. Простіше кажучи, коли два і більше потоків працюють з однією і тією ж змінної, то для того, щоб потоки бачили актуальні дані змінної, вона повинна бути оголошена як volatile.

Декларування змінної як volatile гарантує:

- будь-який з потоків буде бачити найостанніші дані змінної;
- звернення до змінної буде відбуватися у порядку визначеному на стадії компіляції;
- декілька потоків не отримають одночасний доступ до змінної.

Щоб повністю зрозуміти, що значить volatile, по-перше, потрібно зрозуміти, як потоки оперують зі звичайними, не-volatile змінними. З метою підвищення ефективності роботи, специфікація мови Java дозволяє JVM зберігати локальну копію змінної в кожному потоці, який посилається на неї. Можна вважати ці «внутріпоточні» копії змінних схожими на кеш, що

допомагає уникнути перевірки пам'яті кожного разу, коли потрібний доступ до значення змінної. Тепер уявіть, що станеться в наступному випадку: запустяться два потоки, і перший прочитає змінну A як 5, тоді як другий - як 10. Якщо змінна A змінилися від 5 до 10, то перший потік не знатиме про зміну, так що матиме неправильне значення A. Однак якщо змінна A буде позначена як volatile, то в будь-який час, коли потік звертається до її значенню, він буде отримувати копію A і зчитувати її поточне значення.

Алгоритм роботи з volatile змінної дуже схожий з принципом synchronized. За винятком того, що при використанні synchronized робота йде з методами і блоками коду, а при використанні volatile робота йде тільки з однією змінною.

Коли потік звертається до volatile змінної, то відбуваються наступні дії JVM:

1) Встановлюється глобальний lock на змінну.
2) Читається значення змінної з основної пам'яті і записується в локальну змінну потоку.
3) Знімається глобальний lock на змінну.

Розглянемо приклад:

```
public class Checker {
    private volatile int counter = 0;
    public static void main(String[] args)
                                throws InterruptedException {
       Checker obj = new Checker();
       Thread t0 = new Thread(new CounterThread(obj));
       Thread t1 = new Thread(new CounterThread(obj));
        t0.start();
        t1.start();
        t0.join();
        t1.join();
       System.out.println(obj.counter);
    }
    public void increase() {
        counter++;
    }
}

class CounterThread implements Runnable {
    private Checker checker = null;
    public CounterThread(Checker checker) {
        this.checker = checker;
    }
    public void run() {
```

```
            for (int i = 0; i <= 1000000; i++) {
                checker.increase();
            }
        }
    }
```

Два створених потока t1 і t2 отримують доступ до цілого поля counter одного об'єкта obj. Поле позначено ключовим словом volatile і збільшується потоками на одиницю в циклі, який виконується 1000000 разів. Таким чином, очікується, що результат виконання програми буде 2000000. Однак якщо запустити програму, то ви зможете переконатися, що результат роботи буде щоразу різний, але практично завжди менше 2000000. В чому проблема? Справа в тому, що volatile НЕ гарантує атомарность над змінною. Виходить, якщо треба збільшити значення int змінної на одиницю (++), то відбудеться два, а не одне звернення до змінної (counter ++ виконується як counter = counter + 1). Приміром, при тієї ж операції ++, перший потік зчитує volatile змінну з основної пам'яті (поточне значення 1) і оновлює свою локальну змінну (lock -> read -> unlock), потім після розблокування змінної – другий потік зчитує значення змінної (поточне значення 1). Далі, перший потік додає одиницю до значення своєї локальної змінної (поточне значення 2) і записує її в основну пам'ять і потім другий потік додає одиницю до значення своєї локальної змінної (поточне значення 2) і записує її в основну пам'ять. Таким чином, в основній пам'яті буде величина рівна 2, а не 3 як гадалося. Щоб уникнути таких помилок, необхідно використовувати synchronized або готові атомарні класи пакету java.util.concurrent.atomic.

Якщо в прикладі зробити метод синхронізованим

```
public synchronized void increase() {
        counter++;  }
```

то отримаємо вірний результат 2000000.

Зазвичай volatile змінні використовують як прапорці для безлічі потоків. Приміром, для завершення всіх потоків використовують volatile змінну булевского типу в якості глобального прапора.

1.2.3 Переваги концепції монітор

У загальному випадку monitor вирішує наступні проблеми:

– Взаємне виключення (mutual exclusion) – ніякі два і більше активних потоків не можуть перебувати одночасно у синхронізованої по одному об'єкту секції.
– Умовне сподівання (condition waiting) – було реалізовано в обмеженому буфері. Потоки можуть бути переведені в стан очікування методом wait() і пробудитися при виклику методів notify() / notifyAll() при виконанні певної умови.
– Видимість (visibility) – синхронізована секція або метод дозволяють організувати роботу потоків з даними таким чином, щоб будь-який з потоків бачив останні значення змінної.

1.3 Переривання потоків. Методи interrupt() і isInterrupted()

Розглянемо, як можна перервати вже працюючий потік. Подібне завдання може бути актуальним для багатьох програм, в яких необхідно з одного працюючого потоку перервати виконання інших запущених потоків. Наприклад, з потоку браузера після натискання користувачем кнопки Esc призупинити завантаження картинки в HTML сторінку, яке виконується в окремому потоці. Відразу відзначимо, що ця задача нетривіальна і складна.

Для переривання потоків існує кілька методів Java позначених анотацією deprecated, що говорить про те, що методи використовувати небажано і можливо скоро їх видалять з jdk. Методи вважаються небезпечними для використання, так як вони не є в повній мірі "потокобезпечними". До даних методів, що знищують потік, відносяться наступна група методів: destroy(), stop(), suspend() і resume(). Розглянемо їх докладніше в кінці розділу.

Замість deprecated методів були додані інші: interrupt(), isInterrupted() і interrupted().

Розглянемо приклади використання цих методів:

```
public class Example {
public static void main(String[] args)
                      throws InterruptedException{
Thread thread = new Thread (new Runnable(){
   public void run() {
    Thread myThread = Thread.currentThread();
     while (true) {
     System.out.println(myThread.isInterrupted());
      for (long k = 0; k < 1_000_000_000L; k++);
        }
```

```
        }
    });
thread.start();
thread.sleep(1000);
thread.interrupt();
}}
```

Результат виконання:

false
false
true
true
true
true
true
… дуже довго

У програмі в потоці main був створений інший потік thread, який надалі був запущений методом start(). Після запуску потоку thread, потік main спав 1000 мс., а далі для потоку thread був викликаний метод interrupt(). Метод interrupt() переривает виконання потоку. Схематично дана послідовність подій представлена на рис.1.7.

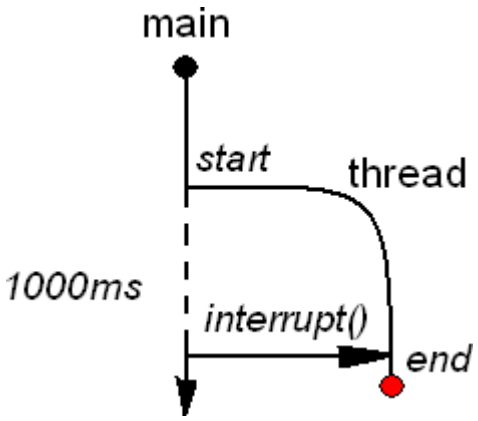

Рисунок 1.7 – Схема переривання потоку методом interrupt()

На відміну від блокуючого методу join() при виклику неблокуючого методу interrupt() не очікується завершення виконання потоку, а він може бути перерваний відразу ж.

Усередині методу run() потоку thread у нескінченному циклі виводиться на екран стан потоку за допомогою нестатичних логічного методу isInterrupted(). Даний метод повертає значення true, якщо потік перерваний методом interrupt() і false в іншому випадку. Таким чином, в Java кожен потік має якийсь булевский прапорець, який при старті потоку

дорівнює false, а при виклику методу interrupt() дорівнює true. Щоб значення прапорця виводилися з деяким інтервалом часу, у програмі використовується додатковий цикл, перебирающий значення змінної-параметра від 0 до 1000000000.

Повернути прапорец переривання потоку можна також за допомогою статичного методу interrupted().

```
public class Example {
     public static void main(String[] args)
                          throws InterruptedException{
     Thread thread = new Thread (new Runnable(){
        public void run() {
          while (true) {
          System.out.println(Thread.interrupted());
           for (long k = 0; k < 1_000_000_000L; k++);
             }
          }
       });
     thread.start();
     thread.sleep(1000);
     thread.interrupt();
     }}
```

Результат виконання:

false
true
false
false
false
false
false
false
false
… дуже довго

Як можна бачити за результатом виконання програми, хоча методи isInterrupted() і interrupted() призначені для одного і того ж, вони мають відмінності. Метод isInterrupted() – не очищає прапор, а метод interrupted () – очищає. Крім того метод isInterrupted() більш вірний з точки зору угоди про іменування і code conventions.

Розглянемо приклад, що демонструє застосування прапора переривання.

```
public class Example {
     public static void main(String[] args)
                          throws InterruptedException{
```

```
     Thread thread = new Thread (new Runnable(){
        public void run() {
        Thread myThread = Thread.currentThread();
          while (!myThread.isInterrupted()) {
          System.out.println("Hello! ");
           for (long k = 0; k < 1_000_000_000L; k++);
             }
          }
       });
     thread.start();
     thread.sleep(1000);
     thread.interrupt();
     }}
```

У методі run() потоку організовано цикл, умовою завершення якого є переривання потоку. Поки потік не перерваний на екран буде виводитися текст Hello!

Зверніть увагу, що метод interrupt() перериває роботу потоку не примусово. Прапор переривання сигналізує про те, що робота потоку перервана, але цей прапор потрібно ще прочитати. Може виникнути ситуація, що потік довго виконує свою роботу, і весь цей час не читає прапор. Тому доцільніше дробити роботу на короткі за часом етапи, і після завершення кожного перевіряти стан прапора переривання. Тому часто використовують інший механізм переривання.

Раніше вже було відзначено, що більшість блокуючих методів (наприклад, метод sleep()) кидає виключення InterruptedException. Якщо прапор переривання потоку встановлений, і він намагається викликати блокуючий метод, то автоматично вилетить виключення InterruptedException, за яким робота потоку завершиться.

Розглянемо наступний приклад:

```
public class Example {
     public static void main(String[] args)
{
     Thread.currentThread().interrupt() ;
     try{
          Thread.sleep(Long.MAX_VALUE);
} catch (InterruptedException e) {
    System.out.println("Interrupted by exception");
}
}}
```

У прикладі потік main сам для себе викликає метод interrupt() за яким встановлюється прапор переривання. Усередині потоку прапор не

перевіряється, потік продовжує виконуватися. Однак коли потік спробує заснути, тобто викликати метод sleep() буде автоматично кинуто виключення InterruptedException, тому здійснюється спроба викликати блокуючий метод при встановленому прапорі переривання. Виконання потоку завершиться. На екрані з'явиться текст Interrupted by exception.

Можна спробувати обдурити JVM – додати метод interrupted(). Даний метод очистить прапор переривання і потік засне.

```
public class Example {
     public static void main(String[] args)
{
     Thread.currentThread().interrupt() ;
     try{
          Thread.interrupted();
          Thread.sleep(Long.MAX_VALUE);
} catch (InterruptedException e) {
    System.out.println("Interrupted by exception");
}
}}
```

Як можна бачити примусово зупинити потік практично не можна.

Повернемося до виключення InterruptedException. Розглянемо ще один варіант, коли можна зупинити потік за допомогою даного винятку. Справа в тому, що якщо потік виконує блокуючий метод (або спить по sleep(), або очікує по wait()) і в цей час для потоку виконується метод interrupt(), то в цьому випадку автоматично кидається виняток InterruptedException. Таким чином, не має значення чи був встановлений прапор переривання до виклику блокуючого методу або пізніше, коли блокуючий метод вже виконувався, в будь-якому випадку вилетить виключення InterruptedException.

```
public class Example {
     public static void main(String[] args)
                          throws InterruptedException{
     Thread thread = new Thread (new Runnable(){
        public void run() {
        try{
        System.out.println("I will sleep");
        Thread.sleep(Long.MAX_VALUE);
        } catch (InterruptedException e) {
          System.out.println("I interrupted by exception");
             }
          }
       });
     thread.start();
```

```
Thread.sleep(1000);
thread.interrupt();
}}
```

Результат:
I will sleep
… через секунду
I interrupted by exception

У прикладі main() стартував новий потік, який заснув. При цьому метод main() продовжив свою роботу, поспав 1 сек., і викликав для запущеного потоку метод interrupt(). Після цього відразу ж вилетить виключення InterruptedException і потік перерветься.

Більш жорстко перервати виконання потоку можна з використанням deprecated методів destroy() і stop(), вони забезпечують інший спосіб переривання. Методи зараз є забороненими до застосування. Так метод destroy() повинен був відразу ж за визначенням вбити потік. У специфікації не було сказано, як завершить роботу потік, чи буде це завершення коректним. Зараз метод destroy() не реалізований. При спробі його використовувати завжди вилітає виключення NoSuchMethodError.

Метод stop() реалізований в Java і вбиває потік більш м'яко, ніж це робив метод destroy(). При виклику методу stop() біля потоку у випадковому місці вилітає unchecked виняток ThreadDeath – спадкоємець Error, що порушує угоду про іменування. Це виключення можна перехопити і коректно завершити роботу потоку.

```
public class Example {
     public static void main(String[] args)
                          throws InterruptedException{
     Thread thread = new Thread (new Runnable(){
        public void run() {
         try{
             while (true) {
          System.out.println("Hello!");
           for (long k = 0; k < 1_000_000_000L; k++);
             }
          } catch (ThreadDeath e) {System.out.println(e);}
         // можно даже продолжить работу потока!!!
         }
       });
     thread.start();
     Thread.sleep(1000);
     thread.stop();
     }}
```

Те, що метод stop() викликає виняток – одне з важливих його переваг перед методом destroy(). Уявіть собі, що потік захопив синхронізовану секцію і його перервали. Якщо його перервали методом stop(), то вилетить виключення і по ньому потік буде викинутий з синхронізованою секції. Якби потік був перерваний методом destroy(), то він би зник моментально, залишивши заблокованою секцію, тобто блокування не була б знята і ніякий інший потік не зміг би увійти в цю синхронізовану секцію.

Ще два deprecated методу – це suspend() і resume(), які не вбивають, а призупиняють потік. За викликом методу suspend() потік призупиняється, а виклик методу resume() знову запускає потік.

```
public class Example {
public static void main(String[] args)
                      throws InterruptedException{
Thread thread = new Thread (new Runnable(){
   public void run() {
        while (true) {
     System.out.println("Hello! ");
      for (long k = 0; k < 1_000_000_000L; k++);
        }
      }
  });
thread.start();
Thread.sleep(3000);
thread.suspend();
Thread.sleep(3000);
thread.resume();
}}
```

Висновок. Метод public void destroy() знищує потік без будь-якої очистки даних, що відносяться до нього, а метод **public final boolean isAlive()** дозволяє визначити, чи запущений потік і ще «живий».

Потоки в Java можуть переривати один одного. Механізм переривань реалізується за допомогою таких методів:

public void interrupt () – перериває даний потік;

public static boolean interrupted () – перевіряє, чи був перерваний даний потік (цей метод очищає ознаку переривання для потоку, тобто при повторному виклику методу для цього ж перерваного потоку він поверне значення false);

public boolean isInterrupted() – аналогічний попередньому методу, але не очищає ознаки переривання для потоку.

1.4 Потоки – демони

Звичайно програма на Java працює до завершення всіх потоків, що входять до неї. Але іноді, зустрічаються потоки, що працюють у фоновому режимі, виконуючи допоміжні дії, які ніколи не закінчуються. Можна помітити такий потік як потік - демон (daemon thread), що говорить JVM про те, що цей потік не треба приймати в розрахунок при визначенні, чи всі потоки даної програми завершилися. Іншими словами, додаток на Java виконується до тих пір, поки не завершиться останній потік, який не є демоном. Потоки, що не помічені як демони, називаються призначеними для користувача потоками (user threads).

Щоб потік вважався демоном, треба скористатися методом **public final void setDaemon(boolean on) throws IllegalThreadStateException**. Якщо параметр on дорівнює true, потік отримує статус демона, якщо false – статус користувальницького потоку. Статус потоку може бути змінений в процесі його виконання. Метод **public final boolean isDaemon()** повертає true, якщо потік є демоном, і false, якщо це користувальницький потік.

Метод **public static int enumerate(Thread[] threadArray)** класу Thread заповнює масив об'єктами Thread, що представляють потоки в групі, до якої належить поточний потік. Оскільки перед таким викликом необхідно створити масив threadArray, треба знати, скільки елементів буде отримано. Метод **public static int activeCount()** класу Thread повідомляє, скільки активних потоків в групі, до якої належить цей потік.

Розглянемо приклад.

```
class T extends Thread {
public void run() {
try {
if (isDaemon()){
 System.out.println("старт потока-демона");
 sleep(10000);
} else {
System.out.println("старт звичайного потока");
}
} catch (InterruptedException e) {
System.err.print("Error" + e);
} finally {
if (!isDaemon())
System.out.println(
"завершення звичайного потока");
else
```

```
System.out.println(
"завершення потока-демона");
}
}
}

public class DemoDaemonThread {
public static void main(String[] args) {
T usual = new T();
T daemon = new T();
daemon.setDaemon(true);
daemon.start();
usual.start();
System.out.println("останній оператор main");
}
}
```

У результаті компіляції і запуску, можливо, буде виведено:

```
останній оператор main
старт потоку-демона
старт звичайного потоку
завершення звичайного потоку
```

Поток-демон (через виклик методу sleep (10000)) не встиг завершити виконання свого коду до завершення основного потоку, пов'язаного з методом main(). Базова властивість потоків-демонів полягає в можливості основного потоку завершити виконання потоку-демона (на відміну від звичайних потоків) із закінченням коду методу main(), не звертаючи уваги на те, що потік-демон ще працює. Якщо зменшувати час затримки потоку-демона, то він може встигнути завершити своє виконання до закінчення роботи основного потоку.

1.5 Пріоритети та групи потоків

Розподіл процесорного часу між потоками в Java виконується за такими правилами: коли потік блокується, тобто припинений, переходить в стан очікування або повинен дочекатися якоїсь події, Java вибирає інший потік з тих, які готові до виконання. Вибирається потік, що має найбільший пріоритет. Якщо таких кілька, вибирається будь-який з них. Пріоритет потоку можна встановити методом public final void setPriority(int newPriority) throws IllegalArgumentException.

Пріоритет потоку повинен бути числом в діапазоні від Thread.MIN_PRIORITY до Thread.MAX_PRIORITY. Будь-яке значення поза

цими межами викликає виключення IllegalArgumentException. За замовчуванням потоку приписується пріоритет Thread.NORM_PRIORITY. Значення пріоритету потоку можна визначити за допомогою методу **public final int getPriority()**.

Клас ThreadGroup реалізує стратегію забезпечення безпеки, яка дозволяє впливати один на одного тільки потокам з однієї групи. Наприклад, потік може змінити пріоритет іншого потоку з тієї ж групи або перевести його в стан очікування. Якби не було розбиття потоків на групи, один потік міг би викликати хаос в середовищі Java, перевівши в стан очікування всі інші потоки, або, що ще гірше, завершивши їх.

Групи потоків організовані в ієрархічну структуру, де у кожної групи є батьківська група. Потоки можуть впливати на потоки зі своєї групи і з дочірніх груп. Групу потоків можна створити, просто задавши її ім'я, за допомогою конструктора **public ThreadGroup (String groupName)**.

Можна також створити групу потоків, дочірню по відношенню до існуючої, використовуючи конструктор **public ThreadGroup(ThreadGroup existingGroup, String groupName) throws NullPointerException**.

Метод **public String toString()** повертає строкове представлення даної групи, а методи **public final String getName()** і **public final void setName(String name)** дозволяють отримати ім'я групи або встановити ім'я групи.

Інші методи класу ThreadGroup є аналогами відповідних методів класу Thread, але застосовуються до всієї групи:

public int activeCount() – повертає число активних потоків в даній групі;

public int activeGroupCount() – повертає число активних груп в даній групі;

public final void checkAccess() – перевіряє, чи може потік, що виконується в даний час, модифікувати дану групу;

public final void destroy() throws IllegalThreadStateException, **SecurityException** і **public boolean isDestroyed()** – відповідно знищує всі потоки даної групи та її підгруп (всі потоки в групі повинні бути попередньо зупинені) або перевіряє чи «жива» дана група;

public final void interrupt() – переривает всі потоки в даній групі;

public final boolean isDaemon() і **public final void setDaemon(boolean daemon)** – відповідно перевіряє і встановлює ознаку потоку-демона для всієї групи.

Можна обмежити пріоритет потоків з групи за допомогою виклику методу **public final synchronized void setMaxPriority(int priority)** , а визначити максимальний пріоритет потоку в групі можна за допомогою **методу public final int getMaxPriority()**.

Батьківська група потоків доступна за допомогою виклику методу **public final ThreadGroup getParent()**, а за допомогою методу **public final boolean parentOf (ThreadGroup g)** можна перевірити чи є група g батьківської для даної групи.

Контрольні питання до розділу 1

1) Які два способи використовуються для реалізації потоків в Java?
2) Як виконується зупинка або завершення потоку?
3) Які операції над потоками визначені в Java?
4) Як виконується переривання потоку і перевірка стану переривання?
5) Що таке потоки-демони і чим вони відрізняються від звичайних потоків?
6) Як задати і отримати значення пріоритету для потоку?
7) Як можна створити групу потоків і які операції визначені для групи потоків в Java?
8) Які засоби синхронізації потоків є в Java?
9) Як виконується синхронізація потоків за допомогою оператора synchronized?
10) Як виконується синхронізація потоків за допомогою методів wait() і notify()?
11) Чим відрізняється робота методу wait() з параметром і без параметра?
12) Чим відрізняються методи Thread.sleep() і Object.wait()?
13) Як працює метод Thread.join()?
14) Що таке dead lock?
15) Як правильно завершити роботу потоку?
16) На якому об'єкті відбувається синхронізація при виклику static synchronized методу?

17) Для чого використовується ключове слово volatile?

18) Яким буде результат виконання наступного програмного кода?

a)
```
public static void main(String[] args)
                          throws InterruptedException{
     Object obj = new Object();
          synchronized (obj) {
               obj.wait();
          }}
```

b)
```
public static void main(String[] args)
                          throws InterruptedException{
     Object obj0 = new Object();
          Object obj = obj0;
          synchronized (obj) {
               obj0.wait();
          }}
```

c)
```
public static void main(String[] args)
                          throws InterruptedException{
          synchronized (new Object) {
               new Object.wait();
          }}
```

2 ПОТОКИ ВВЕДЕННЯ/ВИВЕДЕННЯ І КОЛЕКЦІЇ В JAVA

2.1 Програмування потоків введення/виведення

2.1.1 Робота з файлами в Java

Інформація на пристроях зовнішньої пам'яті зберігається у файлах – іменованих областях на диску, дискеті або іншому машинному носії. Структура (як правило, ієрархічна) розміщення файлів на носіях інформації, а також види та правила завдання атрибутів файлу (імені, типу, дати створення та/або модифікації та інш.) називається файловою системою. Файлові системи в різних операційних системах, як правило, відрізняються.

Клас File. Клас File дозволяє отримати від системи всі дані, починаючи з імені файлу і закінчуючи часом останньої його модифікації. Клас File можна використовувати для створення нових каталогів, а також для видалення і перейменування файлів. Для створення об'єкта File потрібно викликати один з трьох конструкторів класу:

```
File(String path)
File(String path, String name)
File(File dir, String name)
```

Перший конструктор створює об'єкт File із зазначеним повним ім'ям файлу. Другий конструктор створює цей об'єкт, використовуючи окремо шлях і ім'я файлу, а третій створює об'єкт, використовуючи шлях і ім'я файлу, при цьому шлях визначається іншим об'єктом File.

Методи для роботи з файлами і каталогами класу File наведені в таблиці 2.1.

Таблиця 2.1 – Методи для роботи з файлами і каталогами класу File

Метод	Опис
`public String getName()`	Визначає ім'я файлу.
`public String getPath()`	Визначає відносний шлях до файлу (наприклад з даного каталогу).
`public String getAbsolutePath()`	Визначає повне ім'я файлу (шлях до каталогу з кореневого каталогу). Цей шлях називається також абсолютним шляхом до файлу.

Метод	Опис
`public String getParent()`	Визначає батьківський каталог файлу.
`public boolean exists()`	Повертає значення true, якщо файл існує.
`public boolean canWrite()`	Повертає значення true, якщо в файл можна записувати.
`public boolean canRead()`	Повертає значення true, якщо файл можна читати.
`public boolean isFile()`	Повертає значення true, якщо об’єкт є файлом.
`public boolean isDirectory()`	Повертає значення true, якщо об’єкт є каталогом.
`public boolean isAbsolute()`	Повертає значення true, якщо ім'я файлу повне.
`public long lastModified()`	Повертає час останньої модифікації файлу.
`public long length()`	Повертає довжину файлу.
`public boolean mkdir()`	Створює каталог.
`public boolean renameTo (File newName)`	Перейменовує файл.
`public boolean delete()`	Видаляє файл.
`public boolean mkdirs()`	Створює дерево каталогів.
`public String[] list()`	Створює строковий масив імен файлів і підкаталогів в каталозі.
`public String[] list (FilenameFilter filter)`	Створює строковий масив імен всіх файлів, що задовольняють фільтру імен по інтерфейсу FilenameFilter.
`public File[] listFiles()`	Створює масив об'єктів класу File в каталозі.
`public File[] listFiles (FilenameFilter filter)`	Створює масив об'єктів класу File, що задовольняють фільтру імен по інтерфейсу FilenameFilter.
`public File[] listFiles (FileFilter filter)`	Створює строковий масив об'єктів класу File, що задовольняють фільтру імен по інтерфейсу FileFilter.

Часто потрібно обмежити кількість файлів, що повертаються методами list() або listFiles(), щоб включати в список тільки ті з них, імена яких відповідають деякому зразку або фільтру. Для цього можна використовувати перевантажену форму методів list() і listFiles():

```
public String[] list(FilenameFilter filter)
public File[] listFiles(FilenameFilter filter)
```

У цих формах як параметр задається об'єкт класу, який реалізує інтерфейс типу FilenameFilter.

Інтерфейс FilenameFilter визначає єдиний метод accept(), який має наступний вигляд:

```
public abstract boolean accept(File dir, String name)
```

Цей метод повертає true для файлів каталогу dir, які повинні бути включені в відфільтрований список (з іменами name), і повертає false для тих файлів, які повинні бути виключені зі списку (імена яких не збігаються з name).

При використанні методу

```
public File[] listFiles(FileFilter filter)
```

файли фільтруються не просто за іменами, а за їх повними іменам (зі шляхами в ієрархії каталогів). При цьому повертаються файли з тими іменами шляхи, які задовольняють файловому фільтру filter. Інтерфейс FileFilter визначає тільки один метод, accept(), який викликається одного разу для кожного файлу у списку. Його загальна форма:

```
boolean accept(File path)
```

Метод повертає true для файлів, які повинні бути включені в список (тобто тих файлів, які вказані в path), і false – для тих файлів, які повинні бути виключені (не вказані в path).

Змінна public final static char separatorChar класа File повертає символ поділення імен каталогів і файлів ("\" в Windows і "/" – в Unix), а змінна public final static char pathSeparatorChar містить символ-роздільник шляхів у списку (";" в Windows і ":" – в Unix).

Вибір файлів в Swing

Клас JFileChooser в Swing забезпечує діалогове вікно для вибору каталогів і файлів. Основні конструктори цього класу:

```
JFileChooser()
JFileChooser(File currentDirectory)
JFileChooser(String currentDirectoryPath)
```

дозволяють відкрити вікно вибору файлів як для всіх файлів, так і для конкретного каталогу.

Таблиця 2.2 – Методи класу `JFileChooser`

Метод	Опис
`public File getSelectedFile()`	Отримання вибраного файлу.
`public File[] getSelectedFiles()`	Отримання вибраних файлів.
`public void setSelectedFile(File file)`	Установка вибраного файлу.
`public void setSelectedFiles(File[] selectedFiles))`	Установка вибраних файлів.
`public File getCurrentDirectory()`	Отримати вибраний каталог.
`public void setCurrentDirectory(File dir)`	Встановити вибраний каталог.
`public String getDescription(File f)`	Отримання характеристик файлу.
`public void setFileFilter(FileFilter filter)`	Установка фільтра для імен виведених файлів.
`public FileFilter getFileFilter()`	Отримання фільтра для імен виведених файлів.

Для відстеження подій, пов'язаних з кнопками в діалоговому вікні вибору файлів, необхідно реалізувати інтерфейс і додати або видалити блок прослуховування для кнопок діалогового вікна вибору файлів за допомогою методів

```
public void addActionListener(ActionListener l)
```

і

```
public void removeActionListener(ActionListener l)
```

класу `JFileChooser`.

2.1.2 Організація введення-виведення в Java

Всі дані в комп'ютерній системі проходять від пристроїв введення через комп'ютер до пристроїв виводу. Аналогія з перетікаючими даними викликала термін «потоки» (streams). Два терміна в Java, що означають різні поняття: thread і stream, переводяться одним і тим же словом – потік. Щоб уникнути плутанини, будемо використовувати для першого терміна слова «потік обчислень», а для другого – просто «потік».

Потоком називається канал обміну інформацією між її джерелом і одержувачем. На одному кінці потоку завжди знаходиться програма на Java. Якщо вона буде служити джерелом даних, то даний потік буде вихідним, якщо програма буде перебувати на приймаючій стороні – то вхідним.

Починаючи з версії JDK 1.1, в Java визначені два типи потоку: байтовий і символьний потоки.

Байтовий потік або оперує з байтами і використовується при читанні (*input stream*) або запису (*output stream*) даних в двійковому коді.

Символьний потік (*reader* або *writer*) використовується для вводу і виводу символів. У більш старих мовах, наприклад C або C ++, поняття символ і байт є еквівалентними, оскільки символи мали однобайтових кодування. Однак в Java символ не є еквівалентом байта, тому що являє собою 16-бітовий елемент, призначений для розміщення кодів Unicode.

У мові Java потоки представляються класами. Найпростіші з цих класів працюють з базовими потоками вводу і виводу, що мають основні засоби роботи з потоками. Від базових класів породжуються інші класи, більшою мірою орієнтовані на конкретний тип вводу або виводу.

Всі класи вводу/виводу Java описані в пакеті java.io.

Слід зауважити, що практично кожен метод будь-якого класу в пакеті java.io здатний генерувати ту чи іншу форму виняткової ситуації IOException. Тому для поліпшення стилю програмування слід завжди поміщати виклики операцій вводу/виводу до блоку try і catch.

2.1.3 Байтові потоки введення/виведення

Класи InputStream і OutputStream. Абстрактний клас InputStream являє собою базовий потік вводу. Клас містить єдиний конструктор InputStream().

Таблиця 2.3 – Методи класу InputStream

Метод	Опис
`public int read() throws IOException`	Зчитує з вхідного потоку окремі байти як цілі числа, повертаючи значення -1, коли більше нічого читати.
`public int read(byte b[]) throws IOException`	Зчитує безліч байтів в байтовий масив, повертаючи кількість реально введених байтів.
`public int read(byte b[], int off, int len) throws IOException`	Також читає дані в байтовий масив, проте дозволяє вказати зсув (off) в масиві, з якого почнеться запис символів, а також задати максимальне число зчитувальних байтів (len).
`public long skip(long n) throws IOException`	Пропускає байти в потоці.
`public int available() throws IOException`	Повертає кількість байтів, що є в даний момент в потоці.
`public void mark(int readlimit)`	Позначає позицію readlimit в потоці.
`void reset() throws IOException`	Повертається до зазначеної позиції потоку.
`public boolean markSupported()`	Повертає булевське значення, яке вказує на те, чи можна в даному потоці відзначати позиції і повертатися до них.
`public void close() throws IOException`	Закриває потік.

Абстрактний клас OutputStream, забезпечує базові функції для всіх вихідних потоків і має єдиний конструктор OutputStream().

Таблиця 2.4 – Методи класу OutputStream

Метод	Опис
`public void write(int b) throws IOException`	Записує байт в потік.
`public void write(byte b[]) throws IOException`	Записує в потік всі байти, що містяться в заданому байтовому масиві.

`public void write(byte b[], int off, int len) throws IOException`	Записує дані з байтового масиву, вказуючи початковий зсув (off) і кількість виведених байтів (len).
`public void flush() throws IOException`	Виконує примусовий запис всіх буферізованих вихідних даних.

Класи FileInputStream і FileOutputStream. Клас FileInputStream створює (відкриває) об'єкт, який можна використовувати для читання байтів з файлу за допомогою одного з наступних конструкторів:

```
FileInputStream (String filepath) throws FileNotFoundException
FileInputSream(File file) throws FileNotFoundException
```

Клас FileInputStream реалізує методи available(), close(), skip() і всі форми методу read().Крім того, в цьому класі доданий метод

public final FileDescriptor getFD() throws IOException,

який повертає об’єкт FileDescriptor для відкритого файлу і метод

```
protected void finalize() throws IOException
```

для очищення зв'язку з файлом і виклику методу close().

Клас FileOutputStream створює об'єкт OutputStream, який можна застосовувати для запису байтів в файл. Зазвичай використовуються такі конструктори цього класу:

```
FileOutputStream(String filePath)
FileOutputStream(File fileObj)
FileOutputStream(String filePath, boolean append)
```

Параметри в цих конструкторах мають той же зміст, що і в конструкторах класу FileInputStream. Якщо параметр append в останньому конструкторі дорівнює true, файл (якщо він вже існує) відкривається в режимі додавання, інакше файл записується заново.

Клас FileOutputStream реалізує метод close() і всі форми методу write() класу OutputStream, а також використовує свої власні методи getFD() і finalize(), аналогічні однойменним методам класу FileInputStream.

Класи ByteArrayInputStream і ByteArrayOutputStream. Клас ByteArrayInputStream є реалізацією вхідного потоку, яка використовує байтовий масив як джерело. Цей клас має два конструктора, кожен з яких використовує байтовий масив в якості джерела даних:

```
ByteArrayInputStream(byte array[])
```

```
ByteArrayInputStream (byte array [], int off, int len)
```

Тут байтовий масив array – це джерело вводу. Другий конструктор створює об'єкт, що складається з байтового масиву, який починається з позиції off і має довжину len байтів.

Клас ByteArrayInputStream реалізує методи read() класу InputStream для читання оного байта і частини масиву. Реалізація методу reset() в цьому класі має таку особливість: якщо метод mark() не викликало, то reset() встановлює потоковий покажчик на початок потоку, який в цьому випадку є початком масиву байт, переданого конструктору. Цю особливість можна використовувати, наприклад, для читання одного і того ж потоку введення двічі.

При виконанні операції виводу масив байтів можна переслати в локальну пам'ять комп'ютера. Конкретним класом вивідного потоку OutputStream, здатним вирішити це завдання, є ByteArrayOutputStream.

Клас ByteArrayOutputStream має наступні два конструктора:

```
ByteArrayOutputStream()
ByteArrayOutputStream(int len)
```

Першому з них не передається ніяких параметрів, тому він створює буфер розміром 32 байт. При необхідності розмір буфера може бути збільшений. Однак якщо збільшення буде виконуватися на кілька байтів за кожне звернення, то в деяких системах це може викликати сильну фрагментацію пам'яті. Якщо заздалегідь відомо, що буде потрібно буфер розміром, наприклад, в 1024 байта, слід використовувати другу версію конструктора, якому передається один параметр.

До методів, успадкованих від базового класу OutputStream, клас ByteArrayOutputStream ще підтримує такі методи:

Таблиця 2.5 – Методи класу ByteArrayOutputStream

Метод	Опис
`public int size()`	Повертає поточний розмір буфера.
`public void reset()`	Скидає значення лічильника вивідного потоку в 0, так, що вже накопичене вміст буфера вивідного потоку втрачається.
`public byte[] toByteArray()`	Перетворює дані вивідного потоку в новий масив байтів.
`public String toString()`	Перетворює вміст буфера в рядок відповідно до правил кодування символів

	за замовчуванням.
`public void writeTo(OutputStream out) throws IOException`	Записує весь вміст потоку в інший потік.

Клас ByteArrayOutputStream можна використовувати як один з елементів для побудови більш складних об'єктів, включаючи процедури взаємодії між процесами, або для заміни інших потоків (наприклад, потоку мережевих даних) в процесі тестування.

Клас SequenceInputStream. Клас SequenceInputStream дозволяє зчіплювати безліч об'єктів вхідного потоку. Основна форма конструктора цього класу використовує в якості параметрів два об'єкти класу InputStream або його підкласів:

```
SequenceInputStream(InputStream firstStream,
InputStream secondStream)
```

Клас виконує запити читання першого об'єкта типу InputStream (параметр firstStream), поки він не закінчиться, і потім перемикається на другий (параметр secondStream). У свою чергу, використовуючи як одного з об'єктів об'єкт класу SequenceInputStream, можна організувати введення більш ніж двох об'єктів вхідного потоку.

Клас SequenceInputStream реалізує методи available() і close(), а також методи read() для читання байта і частини масиву класу InputStream.

Класи BufferedInputStream і BufferedOutputStream. Байтовий буферизований потік розширює класи фільтрованого потоку, приєднуючи буфер пам'яті до потоків введення/виводу. Такий буфер дозволяє виконувати операції вводу-виводу (використовуючи внутрішній буфер) не з одним, а з декількома байтами одночасно, і, отже, збільшує ефективність роботи програми. При наявності буфера стає можливим такі операції, як пропуск байтів, маркування та перевстановлення потоку. Буферизовані потокові класи – це класи BufferedInputStream і BufferedOutputStream.

Клас BufferedInputStream містить два конструктора:

```
BufferedInputStream (InputStream in)
BufferedInputStream(InputStream in, int size)
```

Перша форма створює буферизований потік, використовуючи розмір буфера, заданий за замовчуванням. У другій формі розмір буфера передається в параметрі size. Змінна

```
protected byte[] buf
```

класу BufferedInputStream містить внутрішній буфер вхідного потоку, а змінна

```
protected int count
```

містить лічильник кількості введених байт (ця величина змінюється від нуля до величини масиву size).

Клас BufferedInputStream підтримує всі методи класу InputStream, за винятком методу read() для читання масиву байт.

Клас BufferedOutputStream є ефективним варіантом класу OutputStream, виконуючим запис байтів у внутрішній буфер, який потім може бути виведений у потік нижнього рівня. Конструктори цього классу

```
BufferedOutputStream(OutputStream out)
BufferedOutputStream(OutputStream out, int size)
```

створюють об'єкт BufferedOutputStream з буфером за замовчуванням (512 байт) або з буфером заданого розміру.

Властивість цього классу

```
protected byte[] buf
```

повертає вміст внутрішнього буфера, а властивість

```
protected int count
```

– кількість виведених у буфер байт. Клас BufferedOutputStream реалізує методи write() запису одного байта і частини масиву байт, а також методи close() і flush() класу OutputStream.

Клас PushbackInputStream. Одне із застосувань буферизації – виконання повернення зчитаного байта назад у вхідний потік (операція pushback). Цю можливість в Java здійснює клас PushbackInputStream. Цей клас має наступні конструктори:

```
PushbackInputStream (InputStream inputStream)
PushbackInputStream( InputStream inputStream, int size)
```

Перша форма створює потоковий об'єкт, який дозволяє повернути один байт у вхідний потік. Друга форма створює потік, який має спеціальний pushback-буфер довжиною size байт. Він дає можливість виконувати повернення декількох байтів у вхідний потік.

Крім методів available(), close(), markSupported() і read() для читання частині масиву байт класу InputStream, в PushbackInputStream визначений метод unread() в наступних формах:

```
void unread(int b)
void unread(byte b[])
void unread(byte b [], int off, int len)
```

Перша форма повертає молодший байт параметра b. Повернений байт буде прочитаний наступним за unread() викликом read(). Друга форма повертає групу байтів – весь буферний масив b[]. Третя форма забеспечує отримання частини масиву b[] (len байт, починаючи з індексу off). Якщо здійснюється спроба повернути байт при заповненому буфері повернення, буде кинуто виключення класу IOException.

Клас PrintStream. Клас PrintStream дозволяє вивести в потік нижнього рівня подання даних примітивного, строкового або об'єктного типу, яке вони матимуть при друку.

Два конструктора класу PrintStream:

```
PrintStream(OutputStream out)
PrintStream(OutputStream out, boolean autoFlush)
```

створюють новий потік для друку. Якщо другий параметр в другому конструкторі дорівнює true, то вивідний буфер буде очищатися в одній з наступних ситуацій: записаний масив байт, викликаний один з методів println() або в вивідний потік записується символ "\ n".

Крім реалізації методів write() запису масиву байт і частини масиву байт, а також методів flush() і close() класу OutputStream, в класі PrintStream визначені також наступні методи:

`protected void setError()` – установка стану помилки для потоку в true;

`public boolean checkError()` – очистка буфера потоку з перевіркою його стану помилки.

Крім цього, в класі визначені два вже відомих методів:

`public void print(`*аргумент*`)`

і `public void println(`*аргумент*`).`

В якості аргументів при виклику цих методів можуть бути задані: `boolean b`, `char c`, `char[] s`, `float f`, `double d`, `int i`, `long l`, `String s` і `Object obj`.

2.1.4 Символьні потоки ввода-виводу

Класи Reader і Writer. Функції класів InputStream і OutputStream для символьних потоків виконують абстрактні класи Reader і Writer.

Клас Reader визначає модель символьного вхідного потоку.

Конструктори класу

```
protected Reader()
protected Reader(Object lock)
```

визначають новий вхідний символьний потік. Критичні секції потоку в першому конструкторі синхронізуються з самим вхідним потоком, а в другому – з вказаним об'єктом lock.

Короткий опис методів класу Reader наведено в табл.2.6.

Таблиця 2.6 – Методи класу Reader

Метод	Опис
`public int read() throws IOException`	Зчитує з вхідного потоку окремі символи як цілі числа, повертаючи значення -1, коли більше нічого читати.
`public int read(char c[]) throws IOException`	Зчитує безліч символів в символьний масив, повертаючи кількість реально введених символів.
`public int read(char c[], int off, int len) throws IOException`	Також читає дані в символьний масив, проте дозволяє вказати зсув (off) в масиві, з якого почнеться запис символів, а також задати максимальне число зчитувальних символів (len).
`public long skip(long n) throws IOException`	Пропускає символи в потоці.
`public boolean ready() throws IOException`	Повертає true, якщо потік готовий до читання, і false - в іншому випадку.
`public void mark(int readlimit)`	Позначає позицію readlimit в потоці.
`public boolean markSupported()`	Повертає булевское значення, яке вказує на те, чи можна в даному потоці відзначати позиції і повертатися до них.

`void reset() throws IOException`	Повертається до зазначеної позиції потоку.
`public void close() throws IOException`	Закриває потік.

Клас Writer визначає модель поточного символьного виводу. Конструктори класу

```
protected Writer()
protected Writer (Object lock)
```

визначають новий вивідний символьний потік аналогічно конструкторам класу Reader.

Методи класу Writer представлені в табл.2.7.

Таблиця 2.7 – Методи класу Writer

Метод	Опис
`public void write(int c) throws IOException`	Записує символ у вихідний потік.
`public void write(char c[]) throws IOException`	Записує у вихідний потік всі символи, що містяться в заданому символьному масиві.
`public void write(char c[], int off, int len) throws IOException`	Записує дані з символьного масиву, вказуючи початковий зсув (off) і кількість виведених символів (len).
`public void write(String str) throws IOException`	Записує рядок у вихідний потік.
`public void write(String str, int off, int len) throws IOException`	Записує дані з рядка, вказуючи початковий зсув (off) і кількість виведених символів (len) рядка.
`public void flush() throws IOException`	Виконує примусовий запис всіх буферізованих вихідних даних.
`public void close() throws IOException`	Закриває вихідний потік.

Класи InputStreamReader і OutputStreamWriter. Класи InputStreamReader і OutputStreamWriter є перехідниками між байтовими і символьними потоками.

Клас InputStreamReader перетворює байтовий потік в символьний потік. Основний конструктор цього класу:

```
InputStreamReader (InputStream in)
```

Цей конструктор створює з байтового потоку in символьний потік відповідно до кодуванням за замовчуванням (кодування за замовчуванням залежить від реалізації віртуальної машини Java і встановлених для неї локальних значень).

Для класу InputStreamReader визначені наступні методи: read() для читання одного символу або частини масиву символів, методи ready() і close() класу Reader.

Клас OutputStreamWriter перетворює символьний потік в байтовий потік. Основний конструктор цього класу:

```
OutputStreamWriter (OutputStream out)
```

Цей конструктор створює з символьного потоку out байтовий потік відповідно до кодуванням за замовчуванням.

Для виведення символів в класі OutputStreamWriter використовуються перші три форми методу write(), метод flush() і метод close() класу Writer.

Класи FileReader і FileWriter. Клас FileReader створює об'єкт, який можна використовувати для читання вмісту файлу. Для створення об'єкта класу FileReader можна використовувати один з наступних конструкторів:

```
FileReader (String fileName) throws FileNotFoundException
FileReader (File file) throws FileNotFoundException
```

У першому конструкторі як параметр задається повне ім'я файлу, а в другому – об'єкт класу File.

Клас FileReader успадковує методи read() для читання символу і частини масиву символів, close() і ready() класу InputStreamReader, а також метод read() для читання масиву символів, mark(), markSupported(), reset() і skip() класу Reader.

Клас FileWriter створює об'єкт, який можна використовувати для запису у файл. Конструктори цього класу:

```
FileWriter (String fileName) throws IOException
FileWriter (File file) throws IOException
```

```
FileWriter (String fileName, boolean append) throws IOException
```

задають в параметрах або повне ім'я файлу (параметр fileName), або об'єкт класу File (параметр file). Другий параметр append останнього конструктора задає режим запису у файл: якщо true, то вивод додається в кінець файлу, якщо false – файл перезаписується.

При створенні об'єкта класу FileWriter, якщо файл існує, він відкривається, якщо файл не існує, він буде створений і відкритий.

Клас FileWriter успадковує перші три форми методу write(), метод flush() і метод close() класу OutputStreamWriter.

Класи CharArrayReader і CharArrayWriter. Клас CharArrayReader є реалізацією вхідного потоку, яка використовує символьний масив як джерело. Для цього класу визначені наступні конструктори:

```
CharArrayReader (char c [])
CharArrayReader (char c [], int off, int len)
```

Перший конструктор створює з масиву символів c вхідний потік, а другий конструктор читає len символів масиву c, починаючи з зміщення off.

Клас реалізує наступні методи класу Reader: skip(), ready(), reset(), mark(), markSupported(), close(), а також методи read() для читання одного символу і частини масиву символів.

Клас CharArrayWriter реалізує вихідний потік, що записує дані в символьний масив. Цей клас має два конструктора:

```
CharArrayWriter ()
CharArrayWriter (int initialSize)
```

У першій формі створюється буфер з розміром, заданим за замовчуванням. У другій формі – буфер з розміром, зазначеним у параметрі initialSize. Буфер міститься в поле buf класу CharArrayWriter. Якщо необхідно, розмір буфера буде збільшено автоматично. Число символів, що зберігаються в буфері, визначається полем count класу CharArrayWriter. І buf і count оголошені з модифікатором protected, тобто є захищеними полями.

Поряд з реалізацією методів write() для запису одного символу, частини масиву символів і частини рядки, flush() і close() класу Writer, в класі CharArrayWriter реалізовані наступні власні методи (табл.2.8).

Клас PrintWriter. Клас PrintWriter виводить форматоване представлення об'єктів в текстовий вивідний потік.

Для класу PrintWriter визначені наступні конструктори:

```
PrintWriter (OutputStream out)
PrintWriter (OutputStream out, boolean autoFlush)
PrintWriter (Writer out)
PrintWriter (Writer out, boolean autoFlush)
```

Таблиця 2.8 – Методи класу CharArrayWriter

Метод	Опис
`public char[] toCharArray()`	Повертає копію даних, що вводяться.
`public String toString()`	Перетворює дані, що вводяться в рядок.
`public int size()`	Повертає поточний розмір буфера.
`public void writeTo(Writer out) throws IOException`	Записує вміст буфера в інший символьний потік.

Перший і другий конструктори створюють новий об'єкт PrintWriter з існуючого вивідного байтового потоку out, третій і четвертий конструктори створюють новий об'єкт PrintWriter з існуючого вивідного символьного потоку out (при цьому створюється проміжний об'єкт OutputStreamWriter, що перетворює символи в байти з використанням кодування символів за замовчуванням). Перший і третій конструктори виводять символи у вихідний потік без автоматичного звільнення буфера при переході на новий рядок. Завдання для autoFlush значення true в другому і четвертому конструкторах викликає очистку буферів при використанні методів println().

Методи цього класу не кидають виключення вводу-виводу. Замість них використовуються методи даного класу

```
protected void setError()
```

і

```
public boolean checkError ()
```

відповідно встановлює стан помилки в true і перевіряє стан помилки.

Методи print() і println() класу PrintWriter мають ті ж аргументи, що і відповідні методи класу PrintStream, і діють аналогічно.

Крім того, для класу PrintWriter визначені всі методи класу Writer.

2.2 Програмування колекцій в Java

2.2.1 Компоненти колекцій

Колекція, або контейнер – це об'єкт, який об'єднує декілька елементів в один об'єкт. Колекції використовуються для зберігання даних, доступу і маніпуляцій з даними, а також для передачі даних від одного методу до іншого. Колекція зазвичай містить дані, які представляють природну групу, наприклад телефонний довідник (колекція відповідностей між ім'ям і телефонним номером). Масив також можна розглядати як колекцію, що об'єднує дані одного типу, елементи якої розташовані послідовно, в порядку зростання індексу.

Всі інтерфейси і класи, що відносяться до колекцій, знаходяться в пакеті java.util.

Схема колекцій (collections framework) – це уніфікована архітектура для представлення і маніпулювання колекціями. Всі схеми колекцій містять наступні три компоненти:

– Інтерфейси – абстрактні типи даних, що представляють колекції. Інтерфейси дозволяють маніпулювати колекціями незалежно від деталей їх подання. В Java, як і в інших об'єктно-орієнтованих мовах, ці інтерфейси зазвичай утворюють ієрархію.

– Реалізації – конкретні реалізації інтерфейсів колекції. За своєю суттю вони є повторно використовуваними структурами даних.

– Алгоритми – методи, що виконують деякі корисні дії, наприклад, пошук або сортування, над об'єктами, що реалізують інтерфейси колекцій. Ці методи називають поліморфними, так як один і той же метод може використовуватися в багатьох різних реалізаціях відповідного інтерфейсу колекцій. По суті алгоритми забезпечують повторно використовувану функціональність.

2.2.2 Интерфейси колекцій

Ключові інтерфейси колекцій введені, починаючи з JDK 1.2, і використовуються для маніпулювання колекціями, а також для передачі їх від одного методу до іншого. Головною метою цих інтерфейсів є можливість маніпулювання колекціями незалежно від деталей їх реалізації.

Інтерфейси утворюють дві ієрархії (рис.2.1).

Інтерфейс Collection є коренем першої ієрархії. Цей інтерфейс представляє групу об'єктів, відомих як його елементи. Деякі реалізації колекцій дозволяють дублювання елементів, інші – ні. Пакет SDK не забезпечує прямий реалізації цього інтерфейсу і він використовується для передачі колекцій і маніпулювання ними, коли потрібна максимальне узагальнення.

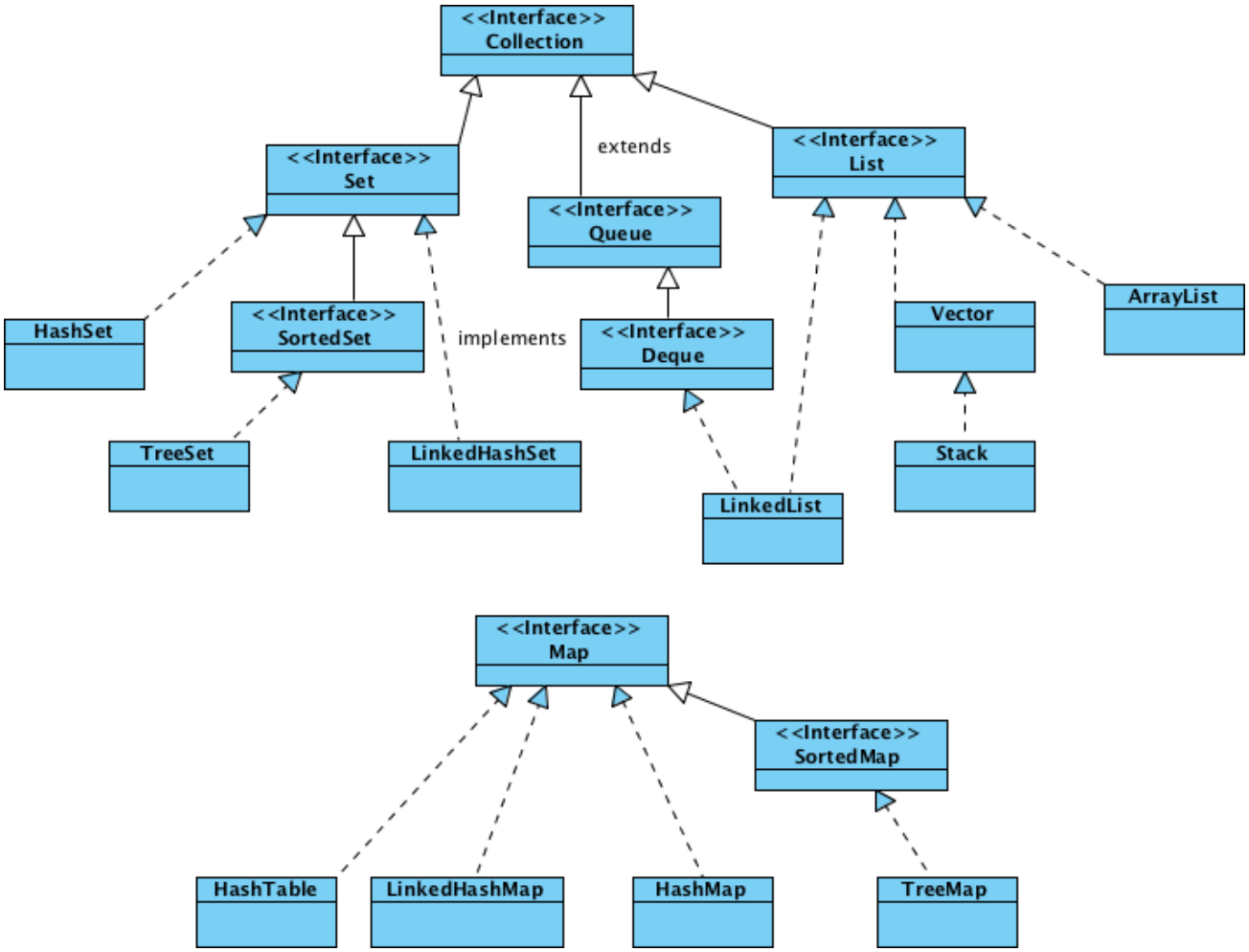

Рисунок 2.1 – Ієрархії інтерфейсів колекцій

Інтерфейс Collection оголошує методи, що виконують такі операції:

– визначення кількості елементів в колекції (методи int size());

– перевірка, чи містить колекція елементи (метод boolean isEmpty());

– перевірка, чи перебуває даний елемент в колекції (метод boolean contains (Object element));

– порівняння заданого об'єкта з даною колекцією на рівність (метод public boolean equals (Object o));

– додавання елемента до колекції (метод boolean add (Object element)) і видалення елемента з колекції (метод boolean remove (Object element)), а також очищення колекції (метод public void clear());

– забезпечення ітераційних операцій над колекцією (метод Iterator iterator());

– забезпечення моста між колекціями і старими інтерфейсами прикладних програм, які припускали передачу їм параметрів об'єкта у вигляді масиву (методи Object[] toArray () і Object[] toArray (Object a[])).

Крім цього, для колекцій визначені наступні групові операції

boolean containsAll (Collection c)

boolean addAll (Collection c)

boolean removeAll (Collection c)

boolean retainAll (Collection c),

які дозволяють перевірити, чи містяться всі елементи колекції с в даній колекції; додати всі елементи колекції с до даної колекції; видалити з даної колекції всі елементи, що містяться в колекції с або залишити в даній колекції тільки ті елементи, які містяться в колекції с.

Метод iterator повертає об'єкт інтерфейсу Iterator. Інтерфейс Iterator оголошує наступні методи для колекцій:

boolean hasNext (),

перевіряє, чи є наступний елемент,

Object next (),

повертає наступний елемент і

void remove (),

видаляє даний елемент.

Нижче наводиться приклад, як об'єкт Iterator використовується для фільтрації колекції, тобто видалення з колекції елементів, що відповідають певним умовам:

```
static void filter(Collection c)
{
        for (Iterator i = c.iterator(); i.hasNext(); )
            if (!cond(i.next()))
                i.remove();
}
```

Даний код є поліморфним, тобто буде працювати для будь-якої колекції, що підтримує видалення елементів, незалежно від реалізації.

Інтерфейс Set є колекцією, яка не повинна містити елементів, що повторюються. Інтерфейс розширює інтерфейс Collection і містить ті ж методи, що і батьківський інтерфейс. Проте методи в цьому інтерфейсі мають більш конкретний математичний сенс. Так, наприклад:

– вираз s1.containsAll (s2) повертає true, якщо s2 є підмножиною s1;

– вираз s1.addAll (s2) перетворює s1 в об'єднання s1 і s2;

– вираз s1.retainAll (s2) перетворює s1 в перетин s1 і s2 (перетин двох множин містить елементи, загальні для обох множин);

– вираз s1.removeAll (s2) видаляє з s1 всі елементи, що містяться в s2.

Інтерфейс List є впорядкованою колекцією (іноді званою послідовністю). Колекція List може містити впорядковані елементи. На додаток до операцій, успадкованих від колекції, інтерфейс оголошує наступні операції:

– отримання значення елемента із заданим індексом (метод public Object get (int index)) і установка значення елемента із заданим індексом (метод public Object set (int index, Object element));

– пошук елемента у списку і повернення значення його індексу (методи public int indexOf (Object o) і public int lastIndexOf (Object o));

– додавання (метод public void add (int index, Object element)) або видалення (метод public Object remove (int index)) елемента по заданому індексу, а також очищення списку (метод public void clear ());

– операції над частиною списку (метод public List subList (int fromIndex, int toIndex));

– ітераційні операції у списку (метод public ListIterator listIterator() і public ListIterator listIterator (int index)).

Інтерфейс ListIterator розширює інтерфейс Iterator і містить наступні методи перегляду списку:

– Object next() – повертає наступний елемент (якщо наступного елемента немає, то викидається виключення типу NoSuchElementException);

– Object previous() – повертає попередній елемент (якщо попереднього елемента немає, то викидається виключення типу NoSuchElementException);

– boolean hasNext() – повертає true, якщо існує наступний елемент, інакше повертає false;

– boolean hasPrevious() – повертає true, якщо існує попередній елемент, інакше повертає false;

– int nextIndex() – повертає індекс наступного елемента (якщо наступного елемента немає, повертає розмір списку);

– int previousIndex() – повертає індекс попереднього елемента (якщо попереднього елемента немає, повертає -1);

– void add (Object obj) – вставляє obj в список перед елементом, який буде повернутий наступним викликом next();

– void remove() – видаляє поточний елемент із списку (викидається виключення типу IllegalStateException, якщо метод remove() викликається, перш ніж викликаний метод next() або previous());

– void set (Object obj) – призначає obj на поточний елемент (це останній елемент, повернутий викликом методу next() або previous()).

Об'єкти в програмах Java упорядковано за допомогою методу

public int compareTo (Object obj),

оголошеного в інтерфейсі Comparable (це єдиний метод даного інтерфейсу). Метод повинен повертати значення 0, якщо порівнювані об'єкти дорівнюють один одному; значення, менше нуля, якщо приймаючий об'єкт менше obj; значення, більше нуля, якщо приймаючий об'єкт більше obj. Конкретні реалізації цього методу в об'єктних розширеннях числових класів (наприклад, Integer або Float) дозволяють порівнювати числа за значенням, для рядків – порівнювати рядки в лексикографічному порядку, для дат – в хронологічному порядку. Сортування, засноване на такому порівнянні, називається природним порядком порівняння.

Але часто необхідно відсортувати елементи колекції в порядку, відмінному від природного або відсортувати елементи без реалізації інтерфейсу Comparable. У цьому випадку необхідно реалізувати інтерфейс Comparator і забезпечити конкретну реалізацію його методів

public int compare (Object o1, Object o2).

і

public boolean equals (Object obj).

Перший метод повертає від'ємне значення, якщо об'єкт o1 менше об'єкта o2, значення 0 у разі рівності об'єктів і додатне значення, якщо об'єкт o1 більше об'єкта o2. Другий метод перевизначає відповідний метод класу Object і перевіряє рівності даного об'єкта інтерфейсу Comparator з об'єктом obj.

Інтерфейс SortedSet є розширенням інтерфейсу Set, елементи якого відсортовані в природному порядку або згідно з порядком, що задається методом інтерфейсу Comparator.

Інтерфейс реалізує наступні операції (додатково до операцій інтерфейсу Set):

– операції над частиною набору (метод public SortedSet subSet (Object fromElement, Object toElement)), головною частиною набору (метод public SortedSet headSet (Object toElement)) і хвостовою частиною набору (метод public SortedSet tailSet (Object fromElement));

– повернення першого (метод public Object first ()) і останнього (метод public Object last ()) елемента відсортованого набору;

– доступ до інтерфейсу Comparator (метод public Comparator comparator()).

Інтерфейс Map є об'єктом, який ставить у відповідність ключам значення. Ключі повинні бути унікальними і їм повинно відповідати єдине значення. Таку колекцію називають відображенням (map), словником (dictionary) або асоціативним масивом (associative array).

В інтерфейсі Map визначені наступні основні операції:

– отримання розміру відображення (метод public int size());

– перевірка відображення на наявність елементів (метод public boolean isEmpty());

– приміщення елемента в відображення (метод public Object put (Object key, Object value)) та отримання значення елемента із заданим ключем (метод public Object get (Object key));

– порівняння відображення з заданим об'єктом на рівність (метод public boolean equals (Object o));

– представлення ключів колекції у вигляді безлічі (метод public Set keySet ());

– видалення елемента із заданим ключем (метод public Object remove (Object key)) і очистка відображення (метод public void clear());

– перевірка наявності елемента із заданим ключем (метод public boolean containsKey (Object key)) або значенням (метод public boolean containsValue (Object value));

– додавання до даного відображенню всіх пар із заданого відображення (метод public void putAll (Map t));

– представлення всіх значень даного відображення у вигляді колекції (метод public Collection values());

– представлення колекції у вигляді безлічі, кожен елемент якого – пара з даного відображення, з якою можна працювати методами вкладеного інтерфейсу Map.Entry (метод public Set entrySet()).

Інтерфейс Map.Entry описує методи роботи з парами «ключ-значення», отриманими методом entrySet():

– отримання ключа (метод public Object getKey()) і значення (метод public Object getValue());

– зміна значення (метод public Object setValue (Object value));

– порівняння заданого об'єкта з даною парою «ключ-значення» на рівність (метод public boolean equals (Object o)).

Можливо також використання об'єктів Map, в яких ключам відповідає кілька значень. Це досягається зазвичай, якщо в якості значень задати об'єкти List.

Інтерфейс SortedMap є розширенням Map. Елементи для цього інтерфейсу відсортовані в природному порядку або згідно з порядком, що задається методами інтерфейсу Comparator.

Інтерфейс реалізує наступні операції (додатково до операцій інтерфейсу Map):

– виділення частини відображення (метод public SortedMap subMap (Object fromKey, Object toKey)), головної частини відображення (метод public SortedMap subMap (Object fromKey, Object toKey)) або хвостовій частині відображення (метод public SortedMap tailMap (Object fromKey));

– отримання першого (метод public Object firstKey()) і останнього (метод public Object lastKey()) елемента відображення;

– доступ до інтерфейсу Comparator (метод public Comparator comparator()).

В цілому, інтерфейс SortedMap є аналогом інтерфейсу SortedSet.

2.2.3 Реалізації колекцій і алгоритми

Реалізації є дійсними об'єктами даних, які втілюють у життя ключові інтерфейси колекцій, описані в попередньому розділі.

Існують три типи реалізацій:

– загальноцільові реалізації;

– пакувальники;

– альтернативні (convenience) реалізації.

Загальноцільові реалізації та їх зв'язок з інтерфейсами і структурами даних представлені в табл.2.9.

Таблиця 2.9 – Реалізації колекцій та їх зв'язок з інтерфейсами і структурами даних

Інтерфейс	Реалізації			
	Хеш-таблиця	Змінюваний масив	Збалансоване дерево	Зв'язаний список
Set	HashSet		TreeSet	LinkedHashSet
List		ArrayList		LinkedList
Map	HashMap WeekHashMap		TreeMap	LinkedHashMap

Всі реалізації мають не тільки схожі імена, але й подібну поведінку. Всі вони реалізують кожну з операцій, визначених у своєму інтерфейсі. Всі дозволяють використовувати елементи, ключі і значення з константою null.

Основним при реалізації структур даних є вибір інтерфейсу. У більшості випадків вибір реалізації впливає тільки на продуктивність програми. Тому переважним стилем програмування є: спочатку створення колекції, потім вибір реалізації і присвоювання нової колекції змінної відповідного інтерфейсного типу (або передача колекції методу, що очікує аргумент интерфейсного типу). Таким чином, програма стає незалежною від будь-яких нових методів, що додаються у даній реалізації, залишаючи за програмістом право зміни реалізації, якщо це необхідно для ефективності програми.

Класи AbstractSet, AbstractList, AbstractSequentialList і AbstractMap. Клас AbstractSet є суперкласом для класів HashSet і TreeSet і містить реалізації методів equals() і removeAll() інтерфейсу Set.

Клас AbstractList є суперкласом для класів AbstractSequentialList, ArrayList, Vector і містить реалізації двох методів add(), методів addAll(), clear(), equals(), get(), indexOf(), iterator(), lastIndexOf(), listIterator(), remove(), set(), sublist() інтерфейсу List.

Клас AbstractList містить також метод

protected void removeRange (int fromIndex, int toIndex)

який дозволяє видалити елементи списку в заданому діапазоні.

Клас AbstractSequentialList є суперкласом для класу LinkedList і містить реалізації методів add(), addAll(), get(), iterator(), listIterator(), remove() і set() інтерфейсу List.

Клас AbstractMap є суперкласом для класів HashMap, TreeMap і WeakHashMap і містить реалізації методів clear(), containsKey(), containsValue(), entrySet(), equals(), get(), isEmpty(), keySet(), put(), putAll(), remove(), size() і values() інтерфейсу Map. Крім того, клас AbstractMap містить метод

protected Object clone()

дозволяє отримати порожню копію відображення (ключі і значення не копіюються), а також метод

public String toString()

дозволяє перетворити відображення в строкове представлення.

Клас HashSet. Для інтерфейсу Set двома загальноцільовими реалізаціями є HashSet і TreeSet.

Клас HashSet реалізує інтерфейс Set с підтримкою хеш-таблиці (зазвичай є реалізацією HashMap) і має наступні конструктори:

– HashSet() – створює новий, порожній набір з ємністю за замовчуванням і фактором завантаження, рівним 0.75;

– HashSet(Collection c) – створює новий набір, що містить елементи заданої колекції;

– HashSet(int initialCapacity) – створює новий порожній набір із заданою ємністю і фактором завантаження, рівним 0.75;

– HashSet(int initialCapacity, float loadFactor) – створює новий порожній набір із заданою ємністю за замовчуванням і заданим чинником завантаження.

У класі HashSet реалізовані наступні методи інтерфейсу Set: add(), clear(), contains(), isEmpty(), iterator(), remove() і size(). Метод

public Object clone()

дозволяє отримати порожню копію об'єкта HashSet (без елементів).

Клас ArrayList. Клас ArrayList забезпечує реалізацію інтерфейсу Set для масивів із змінними розмірами. Так само, як об'єкти класу StringBuffer, об'єкти ArrayList мають дві характеристики – ємність і розмір масиву. Якщо

розмір списку перевищить ємність, ємність автоматично збільшується на деяку кількість елементів.

Клас ArrayList має наступні конструктори:

– ArrayList() – створює порожній список;

– ArrayList(Collection c) – створює список із заданої колекції в порядку, заданому ітератором даної колекції;

– ArrayList(int initialCapacity) – створює порожній список із заданою ємністю.

Клас ArrayList реалізує наступні методи інтерфейсу List: два методи add(), два методи addAll(), методи clear(), contains(), get(), indexOf(), isEmpty(), lastIndexOf(), remove(), set(), size() і два методи toArray().

Крім того, клас містить наступні власні методи:

– public Object clone() – отримання порожній копії об'єкта ArrayList (без елементів);

– public void ensureCapacity (int minCapacity) – збільшує ємність екземпляру ArrayList, якщо це необхідно, для того, щоб він міг містити число елементів, задане в minCapacity;

– protected void removeRange (int fromIndex, int toIndex) – видаляє елементи в заданому діапазоні індексів;

– public void trimToSize() – зменшує ємність об'єкта ArrayList до його попереднього значення.

Клас LinkedList. Клас LinkedList, на додаток до реалізації методів інтерфейсу List забезпечує методи для отримання, видалення і вставки елементів в список, що дозволяє використовувати зв'язані списки як стеки, черги або черги з двома кінцями.

Клас LinkedList має два конструктора:

– LinkedList () – створює порожній зв'язаний список;

– LinkedList (Collection c) – створює зв'язаний список із заданої колекції в порядку, заданому ітератором даної колекції.

На додаток до двох методів add(), двох методів addAll(), методів clear(), contains(), get(), indexOf(), isEmpty(), lastIndexOf(), listIterator(), двох методів remove(), методів set(), size() і двох методів toArray() інтерфейсу List, клас LinkedList забезпечує наступні методи:

– public void addFirst (Object o) і public void addLast (Object o) – додавання елемента в початок або кінець зв'язаного списку;

– public Object getFirst() і public Object getLast() – отримання першого або останнього елемента зв'язаного списку;

– public Object removeFirst() і public Object removeLast() – видалення першого або останнього елемента зв'язаного списку;

– public Object clone() – отримання порожній копії об'єкта LinkedList (без елементів).

Клас ArrayList є кращим перед LinkedList у випадку, якщо необхідна висока продуктивність. Однак, якщо часто доводиться додавати елементи в початок списку або видаляти елементи з його середини, кращим є клас LinkedList. Цей клас повністю реалізує всі можливості класу Stack.

Клас HashMap. Клас HashMap є реалізацією інтерфейсу Map з використанням хеш-таблиць і за своїми можливостями відповідає класу HashTable.

Клас HashMap має наступні конструктори:

– HashMap() – створює нове, пусте відображення з ємністю за замовчуванням 16 і фактором завантаження, рівним 0.75;

– HashMap (Map m) – створює нове відображення, що містить елементи заданого відображення;

– HashMap (int initialCapacity) – створює нове, пусте відображення із заданою ємністю і фактором завантаження, рівним 0.75;

– HashMap (int initialCapacity, float loadFactor) – створює нове, пусте відображення із заданою ємністю і заданим фактором завантаження.

– Клас HashMap реалізує наступні методи інтерфейсу Map: clear(), containsKey(), containsValue(), entrySet(), get(), isEmpty(), keySet(), put(), putAll(), remove(), size() і values().

Клас Collections. Реалізації-оболонки делегують всю свою роботу колекціям, але додають в колекції деяку функціональність. Ці реалізації є анонімними, тобто вони не забезпечують загальнодоступний клас, а реалізуються за допомогою статичних методів в класі Collections.

Клас Collections оперує з колекціями або повертає колекції. Клас не містить конструктора, а тільки такі поля з модифікаторами public static final:

List EMPTY_LIST для порожніх списків;

Map EMPTY_MAP для порожніх відображень

Set EMPTY_SET для порожніх множин, а також наступні public static методи, в основному реалізують поліморфні алгоритми:

– int binarySearch (List list, Object key) – двійковий пошук в заданому списку заданого об'єкта;

– int binarySearch (List list, Object key, Comparator c) – двійковий пошук в заданому списку заданого об'єкта з використанням компаратора;

– void copy (List dest, List src) – копіювання списку-джерела в список призначення;

– void fill (List dest, Object o) – заповнення списку заданих об'єктом;

– Object max (Collection c) і Object min (Collection c) – повернення максимального або мінімального елемента колекції відповідно до природного порядку порівняння;

– Object max (Collection c, Comparator com) і Object min (Collection c, Comparator com) – повернення максимального або мінімального елемента колекції відповідно до порядку, заданих за допомогою компаратора;

– void reverse (List l) – переставляє елементи списку у зворотному порядку;

– void shuffle (List l) – перемішує елементи списку з використанням генератора випадкових чисел за замовчанням;

– void shuffle (List l, Random rnd) – перемішує елементи списку з використанням заданого генератора випадкових чисел;

– List singletonList (Object o) – повертає список, який містить лише даний об'єкт;

– Map singletonMap (Object key, Object value) – повертає відображення, що містить тільки даний ключ і дане значення;

– void sort (List l) і void sort (List l, Comparator c) – сортує список по зростанню відповідно до природнього порядку або з використанням компаратора;

– Collection synchronizedCollection (Collection c), List synchronizedList (List l), Map synchronizedMap (Map m), Set synchronizedSet (Set s), SortedSet synchronizedSortedSet (SortedSet ss) і SortedMap synchronizedSortedMap (SortedMap sm) – створюють синхронізовані (які не залежать від потоків) екземпляри відповідних типів колекцій;

– Collection unmodifiableCollection (Collection c), List unmodifiableList (List l), Map unmodifiableMap (Map m), Set unmodifiableSet (Set s), SortedSet

unmodifiableSortedSet (SortedSet ss) і SortedMap unmodifiableSortedMap (SortedMap sm) – створюють незмінні екземпляри відповідних типів колекцій.

Контрольні питання до розділу 2

1) З якою метою використовується клас File? Які операції над файлами визначені в класі File?
2) Як в класі File реалізована фільтрація файлів?
3) Які засоби для навігації по файлах і каталогах забезпечує клас JFileChooser?
4) Що таке потік даних? Які види потоків визначені в Java?
5) Які методи читання і запису даних визначені в класах InputStream і OutputStream?
6) Як в Java реалізований байтовий обмін даними з файлами?
7) З якою метою використовуються класи ByteArrayInputStream і ByteArrayOutputStream?
8) Як в Java виконується об'єднання двох потоків вводу в один потік?
9) Які засоби буферизації потоків введення-виведення визначені в Java?
10) Як в Java виконується виведення байтових і символьних потоків на друк?
11) Які методи читання і запису даних визначені в класах Reader і Writer?
12) Як в Java реалізований перехід між байтовими і символьними потоками?
13) Як в Java реалізований символьний обмін даними з файлами?
14) Як в Java реалізований введення-виведення в символьні масиви?
15) З якою метою використовується клас StreamTokenizer?
16) Які методи аналізу даних визначено в класі StreamTokenizer?
17) Як визначаються колекції в мові Java? Які компоненти містить схема колекцій в Java?
18) Які інтерфейси колекцій визначені в мові Java?
19) Які операції над колекціями визначені за допомогою методів інтерфейсу Collection?
20) Які методи для перегляду елементів колекцій визначені в інтерфейсі Iterator?
21) Які додаткові методи для колекцій визначені в інтерфейсі List?

22) Які методи перегляду списку містить інтерфейс ListIterator?
23) Як упорядковано об'єкти з використанням інтерфейсу Comparable в програмах на мові Java?
24) Які методи (додатково до методів інтерфейсу Set) реалізує інтерфейс SortedSet?
25) Як визначається інтерфейс Map, і які операції визначені в цьому інтерфейсі?
26) Які методи (додатково до методів інтерфейсу Map) реалізує інтерфейс SortedMap?
27) Які типи реалізацій колекцій існують у мові Java? Дайте коротку характеристику кожного типу.
28) Які абстрактні класи реалізації колекцій визначені в Java? Дайте коротку характеристику кожного класу.
29) Як за допомогою класу ArrayList реалізуються масиви із змінними розмірами в програмах на мові Java?
30) Як в класі LinkedList реалізуються пов'язані списки?
31) Як в класі HashMap в Java реалізуються відображення з підтримкою хеш-таблиць?
32) Які операції над колекціями забезпечують методи класу Collections?

3 РОЗРОБКА МЕРЕЖЕВИХ ПРОГРАМ

При роботі в мережі з застосуванням класів і методів пакету **java.net** будемо використовувати основні поняття, що характеризують мережеві з'єднання: IP - адресація, доменна служба імен, URL адреси, протоколи TCP і UDP, прикладні протоколи, сокети. У зв'язку з цим рекомендується повторити згадані теми за конспектом лекцій з дисципліни «Комп'ютерні мережі».

У пакеті **java.net** визначена група класів та інтерфейсів, що дозволяють управляти мережевими з'єднаннями в Java-програмах. Ось деякі з завдань, які можуть бути вирішені засобами **java.net**:

– Управління адресацією;

– Організація високорівневого програмування з використанням методів класу URL;

– Розробка додатків клієнт-сервер і управління мережевими з'єднаннями на рівні сокетів;

– Управління з'єднаннями на рівні дейтаграм.

Розглянемо ці завдання більш докладно.

3.1 Клас InetAddress

Клас **InetAddress** інкапсулює як числову IP - адресу, так і доменну адресу.

Клас **InetAddress** не має видимих конструкторів. Для створення об'єкта потрібно використовувати один з доступних фабричних методів.

Фабричні методи – просто угода, за допомогою якого статичні методи в класі повертають екземпляр даного класу. Це зроблено замість перевантаження конструктора різними списками параметрів, коли наявність унікальних імен методів призводить до більш ясних результатів.

Для створення об'єктів InetAddress можна використовувати три методи:

– **static InetAddress getLocalHost() trows UnknownHostException** – повертає об'єкт класу InetAddress для локального комп'ютера.

– **static InetAddress getByName (String hostName) trows UnknownHostException** – повертає об'єкт класу InetAddress для комп'ютера, DNS - ім'я якого передається в параметрі hostName.

– **static InetAddress[] getAllByName (String hostName) trows UnknownHostException** – повертає масив об'єктів класу InetAddress, що представляє всі адреси, пов'язані з зазначеним DNS-ім'ям hostName. Використовується в тому випадку, коли групі IP - адрес відповідає одне DNS-ім'я. Це дає можливість зменшити навантаження на найбільш часто відвідувані сайти.

Всі ці методи в разі неможливості розпізнавання імені комп'ютера викидають виключення типу **UnknownHostException**.

Таблиця 3.1 – Основні нестатичні методи класу **InetAddress**

Метод	Опис
`boolean equals(Object other)`	Повертає true, якщо повертаємий об'єкт має ту же IP - адресу, що і об'єкт, отриманий через параметр other
`byte[] getAddress()`	Повертає чотирьохелементний масив байтів, який представляє IP - адресу оброблюваного об'єкта
`String getHostAddress()`	Повертає рядок, який представляє собою адресу комп'ютера, пов'язаного з об'єктом класу InetAddress
`String getHostName()`	Повертає рядок, який представляє собою ім'я комп'ютера, пов'язаного з об'єктом класу InetAddress
`int hashCode()`	Повертає хеш-код викликаємого об'єкта
`boolean isMulticastAddress()`	Повертає true, якщо IP - адреса об'єкта цього класу – групова (multicast).
`String toString()`	Повертає рядок, який перераховує доменну і IP - адресу хост-комп'ютера (наприклад, "sterwave.com/192.147.170.6")
`boolean equals(Object other)`	Повертає true, якщо повертаємий об'єкт має ту же IP - адресу, що і об'єкт, отриманий через параметр other

Приклад. Розглянемо різні випадки застосування розглянутих раніше методів класу **InetAddress**, для отримання IP - адрес.

Зверніть увагу на одне розходження при використанні методів **getByName()** з параметром null і **getLocalHost()**. Якщо локальна мережа побудована на протоколі TCP/IP, то кожному комп'ютеру присвоюється IP - адреса. Тому метод **getLocalHost()** повертає дану IP - адресу цієї локальної

мережі. З іншого боку метод **getByName()** з параметром null в будь-якому випадку повертає адресу 27.0.0.1.

```
import java.net.*;
public class My{

public static void main(String[] args)
throws UnknownHostException {

InetAddress localaddr = InetAddress.getLocalHost();
System.out.println("Local
                    address:"+localaddr.getHostAddress()+
                              "             |             "+
localaddr.getHostName());

localaddr = InetAddress.getByName(null);
System.out.println("Local
                    address:"+localaddr.getHostAddress()+
                              "             |             "+
localaddr.getHostName());

InetAddress addr = InetAddress.getByName("www.yandex.ru");
System.out.println("Yandex address:"+addr.getHostAddress()+
                              " | "+ addr.getHostName());

InetAddress multaddr[] =
InetAddress.getAllByName("www.google.com");
System.out.println("Google address:");
for(int i=0; i<multaddr.length; i++)
System.out.println((i+1)+": "+multaddr[i].getHostAddress()+
                             "             |             "+
multaddr[i].getHostName());
}}
```

Результат роботи:

```
Local address:169.254.86.153 | SERVER
Local address:127.0.0.1   | localhost
Yandex address:77.88.21.3 | www.yandex.ru
Google address:
1: 209.85.129.104 | www.google.com
2: 209.85.129.147 | www.google.com
3: 209.85.129.99  | www.google.com
```

3.2 Класи URL і URLConnection

URL забезпечує зручну форму ідентифікації ресурсів мережі Internet і може складатися з 4-х складових: протоколу, доменного імені або IP - адреси комп'ютера, номера порту і шляху до файлу.

В якості протоколу можуть використовуватися http, ftp, gopher і file. Ім'я протоколу в URL завершується двокрапкою, потім після подвійного слеш (//) вказується DNS-ім'я комп'ютера або IP-адреса, наприклад http://www.yandex.ru/ або http://77.88.21.3/

Часто на комп'ютері в мережі відкривається група портів, через які відбуваються з'єднання. Для того, щоб вказати порт підключення, необхідно ввести його значення після імені комп'ютера, де як роздільник використовується двокрапка: http://somewhere.org:8888/

За ім'ям комп'ютера і номерів порту в URL адресі можна вказати шлях до файлу, для цього як роздільник використовують одинарний слеш (/): http://java.sun.com/docs/index.html

Для використання адрес URL в пакеті **java.net** визначено клас **URL**. Конструктори:

URL (String spec);

URL (String protocol, String host, String file);

URL (String protocol, String host, int port, String file);

де spec – строковий еквівалент адреси URL,

protocol – протокол,

host – ім'я комп'ютера даної URL - адреси,

port – номер порта підключення,

file – ім'я файлу, що завантажується.

Якщо вказана невірна URL – адреса, то буде згенеровано виключення **MalformedURLException**.

Методи класу **URL**

getProtocol(), getHost(), getPort(), getFile() – повертають значення протоколу, імені комп'ютера, номера порту і шляху до файлу відповідно.

openStream() – відкриває потік вхідних даних **InputStream** і дозволяє здійснити завантаження об'єкта класу **URL**.

Приклад. Завантажимо на екран вміст Web-сторінки, розташованої за адресою **http://www.yahoo.com/**

```
import java.net.*;
import java.io.*;

public class My{
public static void main(String[] args) {
try{
URL myURL = new URL("http://www.yahoo.com/");
// Создаем поток ввода
InputStream in = myURL.openStream();
```

```
// Читаем из потока и выводим на экран
int ch;
while((ch=in.read())!=-1)
System.out.print( (char)ch);
in.close();
}
// обработка исключений
catch(MalformedURLException me)
{System.out.println(me.getMessage());
 System.exit(0); }

catch(IOException e)
{System.out.println(e.getMessage());
 System.exit(0); }
}}
```

Для реалізації процесу завантаження інформації з мережі згідно об'єкту **URL** в пакеті **java.net** оголошений відповідний абстрактний клас **URLConnection**. У ньому зібрані методи, що керують процесом завантаження і його інформаційною підтримкою. Створити об'єкт **URLConnection** можна, використовуючи метод **openConnection()** класу **URL**.

URL myURL = new URL ("http://www.yahoo.com/");

URLConnection myCon = myURL.openConnection ();

Після цього буде доступний набір методів, які керують процесом завантаження. Ось деякі з них:

connect() – виконує з'єднання;

getContentLength() – повертає розмір завантаження об'єкта;

getContentType() – повертає тип вмісту об'єкта;

getDate() – повертає дату завантаження;

getExpiration() – повертає термін зберігання;

getPermission() – визначає параметри доступу;

getInputStream() – повертає потік введення InputStream завантажуваних даних.

3.3 Використання сокетів у розподілених додатках

Сокети – це мережеві роз'єми, через які здійснюються двонаправлені потокові з'єднання між комп'ютерами. Сокет визначається номером порту та IP-адресою. При цьому IP-адреса використовується для ідентифікації комп'ютера, номер порту – для ідентифікації процесу, що працює на комп'ютері. Коли один додаток знає сокет іншого, створюється сокетне

з'єднання. Клієнт намагається з'єднатися з сервером, ініціалізувавши сокетне з'єднання. Сервер чекає, поки клієнт зв'яжеться з ним. Перше повідомлення, що посилається клієнтом на сервер, містить сокет клієнта. Сервер в свою чергу створює сокет, який буде використовуватися для зв'язку з клієнтом, і посилає його клієнту з першим повідомленням. Після цього встановлюється комунікаційне з'єднання.

У Java в пакеті **java.net** визначені два класи **ServerSocket** і **Socket**, які реалізують програмування TCP/IP сокетів. Клас **ServerSocket** визначає серверне з'єднання і виконує функції прослуховування заданого порту, чекаючи з'єднання з клієнтом. З іншого боку, мережевий сокет **Socket** виконує функції клієнтського з'єднання, з його допомогою реалізується підключення до заданого адресою серверу, ініціалізація різних протоколів і передача або одержання даних.

При створенні об'єкта класу **Socket** вказується IP-адреса сервера і номер порту (80 для HTTP). Якщо вказано ім'я домену, то Java перетворить його за допомогою DNS-сервера до IP-адреси:

```
try {
Socket socket = new Socket("localhost", 8030);
} catch (IOException e) {
System.out.println("ошибка: " + e);
}
```

Сервер очікує повідомлення клієнта і повинен бути запущений із зазначенням певного порту. Об'єкт класу **ServerSocket** створюється конструктором із зазначенням номера порту і очікує повідомлення клієнта за допомогою методу **accept()**, який є блокуючим, тобто він буде чекати клієнта, щоб ініціалізувати зв'язок, і потім поверне Socket-об'єкт, який буде використовуватися для зв'язку з клієнтом:

```
try {
Socket s = null;
ServerSocket server = new ServerSocket(8030);
s = server.accept();
} catch (IOException e) {
System.out.println("ошибка: " + e);
}
```

Важливо розуміти, що клас **ServerSocket** визначає тільки процес прослуховування певного порту, а об'єкт Socket – процес передачі даних через нього.

Клієнт і сервер після встановлення сокетного зв’язку можуть отримувати дані з потоку введення і записувати дані в потік виводу за допомогою методів getInputStream() і getOutputStream() або PrintStream для того, щоб програма могла трактувати потік як вихідні файли.

Для коректної роботи мережевих додатків, відкритих з використанням об'єктів **ServerSocket** і **Socket,** сокети в кінці роботи з ними необхідно закрити, використовуючи методи **close()**, оголошені в кожному з цих класів.

3.3.1 Організація серверного сокета

Таблиця 3.2 – Конструктори класу ServerSocket

Конструктор	Опис
`ServerSocket(int port)`	Створює сокет сервера на зазначеному порту з довжиною черги за замовчуванням (50)
`ServerSocket(int port, int maxQueue)`	Створює сокет сервера на зазначеному порту з максимальною довжиною черги **maxQueue**[2]
`ServerSocket(int port, int maxQueue, InetAddress localAddress)`	Створює сокет сервера на зазначеному порту з максимальною довжиною черги **maxQueue.** На груповому[3] хост-комп'ютері localAddress визначає IP-адресу, з якою цей сокет пов'язаний.

Приклад. Сервер для прослуховування заданого порту і передачі через нього даних клієнта. У даному прикладі сервер посилає клієнтові рядок "привіт!", після чого розриває зв'язок.

```
import java.io.*;
import java.net.*;

public class MyServer{

public static void main(String[] args)throws Exception {
Socket s = null;
try { //посылка строки клиенту
ServerSocket server = new ServerSocket(2000);
System.out.println("Server running...");
s = server.accept();
PrintStream ps = new PrintStream(s.getOutputStream());
```

[2] Довжина черги повідомляє системі, скільки підключень клієнта вона може залишати затриманими перш, ніж повинна буде просто відкинути з'єднання. За замовчуванням - 50

[3] Груповий хост – комп'ютер, який має декілька мережевих адаптерів або налаштований з декількома IP-адресами для одиночного мережевого адаптера.

```
ps.println("привет!");
ps.flush();//Освобождает буффер
s.close(); // разрыв соединения
} catch (IOException e) {
System.out.println("ошибка: " + e);
}
}}
```

3.3.2 Організація клієнтського сокета

Таблиця 3.3 – Методи і конструктори класу Socket

Конструктор/Метод	Опис
`Socket(String hostName, int port)`	Створює сокет, що з'єднує локальну хост-машину з іменованною хост-машиною і портом; може викидати виключення UnknownHostException або IOException
`Socket(InetAddress ipAddress,int port)`	Створює сокет, аналогічний попередньому, але використовується вже існуючий об'єкт класу InetAddress і порт; може викидати виключення IOException
`InetAddress getInetAddress()`	Повертає InetAddress-об'єкт, пов'язаний з Socket - об'єктом
`int getPort()`	Повертає віддалений порт, з яким з'єднаний даний Socket - об'єкт
`int getLocalPort()`	Повертає локальний порт, з яким з'єднаний даний Socket - об'єкт
`InetAddress getLocalAddress()`	Повертає адресу комп'ютера-клієнта.

При роботі з об'єктами **Socket** можуть виникати виняткові ситуації, для чого потрібно буде реалізувати відповідну обробку наступних виключень:

IOException – помилка вводу/виводу потоків;

SecurityException – помилка доступу до системи безпеки (якщо вона визначена на комп'ютері);

UnknownHostException – неправильно вказана адреса InetAddress.

Приклад. Програмування клієнта, який підключається до створеного раніше додатком серверу, зчитує дані і виводить їх на екран.

```
import java.io.*;
import java.net.*;
```

```
public class MyClient {

public static void main(String[] args) {

System.out.println("Client running...");
try {//получение строки клиентом

//Определяем адрес и порт подключения. Тестируем клиента и
сервера на одном //компьютере
InetAddress local = InetAddress.getLocalHost();
int port = 2000;

//Выполняем подключение
Socket socket = new Socket(local, port);

//Считываем поток данных по сети
BufferedReader dis = new BufferedReader(
new InputStreamReader(socket.getInputStream()));
String msg = dis.readLine();
System.out.println(msg);
socket.close();
dis.close();
} catch (IOException e) {
System.out.println("ошибка: " + e);
}
}
}
```

Аналогічно клієнт може послати дані серверу через потік виводу за допомогою методу **getOutputStream()**, а сервер може отримувати дані за допомогою методу **getInputStream()**.

3.3.3 Багатопоточність у мережевих застосуваннях

Сервер повинен підтримувати багатопоточність, інакше він буде не в змозі обробляти кілька з'єднань одночасно. Сервер містить цикл, що очікує нового клієнтського з'єднання. Кожен раз, коли клієнт просить з'єднання, сервер створює новий потік. У наступному прикладі створюється клас **NetServerThread**, що розширює клас **Thread**. Сервер передає кожному новому підключеному клієнту його порядковий номер.

```
import java.net.*;
import java.io.*;
public class NetServerThread extends Thread {
Socket socket;
int nom; //номер подключения
PrintStream ps;
String msg;
```

```
public NetServerThread(Socket s, int nom) {
socket = s; this.nom=nom;
try {
ps = new PrintStream(s.getOutputStream());
} catch (IOException e) {
System.out.println("ошибка: " + e);
}
}
public static void main(String[] args) {
try {
ServerSocket server = new ServerSocket(2000);
System.out.println("Server running...");
int i=0;//счетчик подключений
while (true) {
Socket s = null;
s = server.accept(); i++;
NetServerThread nst = new NetServerThread(s, i);
nst.start();
}
} catch (IOException e) {
System.out.println("ошибка: " + e);
}
}
public void run() {
String msg = "сообщение: " + nom;
sendMessage(msg);
}

public void sendMessage(String msg) {
ps.println(msg);
System.out.println(msg + "<передача>");
ps.flush();}}
```

Для клієнтських додатків підтримка багатопоточності також необхідна. Наприклад, один потік очікує виконання операції вводу/виводу, а інші потоки виконують свої функції. Нижче наведено приклад програми, що реалізує отримання повідомлення клієнтом в потоці.

```
import java.net.*;
import java.io.*;

public class NetClientThread extends Thread {
BufferedReader br = null;
Socket s = null;

public NetClientThread() {
try {//соединение с кольцевым адресом
s = new Socket("127.0.0.1", 2000);
InputStreamReader isr =
new InputStreamReader (s.getInputStream());
```

```
br = new BufferedReader(isr);
} catch (IOException e) {
System.out.println("ошибка: " + e);
}}

public static void main(String[] args) {
NetClientThread nct = new NetClientThread();
nct.start();
}

public void run() {
while (true) {
try {
String msg = br.readLine();
if (msg == null) break;
else System.out.println(msg);
} catch (IOException e) {
System.out.println("ошибка: " + e);
}}}}
```

Не забудьте, що сервер повинен бути створений до того, як клієнт спробує здійснити сокетне з'єднання. При цьому може бути використана IP-адреса локального комп'ютера.

3.4 Датаграми і протокол UDP

Дейтаграми – невеликі за обсягом пакети даних, які передаються між комп'ютерами по мережі на основі протоколу UDP (User Datagram Protocol). Для роботи з дейтаграммами в пакеті java.net існують два класи **DatagramSocket** і **DatagramPacket**. Об'єкт **DatagramSocket** створює сам сокет, а дейтаграма являє собою об'єкт класу **DatagramPacket**.

У класі **DatagramSocket** є наступні конструктори:

DatagramSocket ();

DatagramSocket (int port);

DatagramSocket (int port, InetAddress addr);

де **port** – номер порту, **addr** – адреса підключення.

Методами **send()** і **receive()**, можна виконувати відповідно відправку та отримання пакетів **DatagramPacket**. Однак попередньо потрібно оголосити екземпляри класу **DatagramPacket**, так як методи **send()** і **receive()** приймають параметр – об'єкт даного класу.

Таблиця 3.4 – Методи класу DatagramSocket

Метод	Опис
`getInetAddress()`	Повертає адресу, до якої здійснюється підключення
`getPort()`	Повертає порт, до якого здійснюється підключення
`getLocalAddress()`	Повертає локальну адресу комп'ютера, з якої виконується підключення
`getLocalPort()`	Повертає локальний порт, через який здійснюється підключення
`receive(DatagramPacket)`	Чекає отримання дейтаграми і копіює дані в спеціальний DatagramPacket
`send(DatagramPacket)`	Посилає DatagramPacket
`close()`	Закриває сокет

Конструктори класу **DatagramPacket**

для отримання пакета

DatagramPacket (byte [] buf, int length);

для відправки пакету

DatagramPacket (byte [] buf, int length, InetAddress address, int port);

де **buf** – масив байт, що представляють пакет дейтаграми,

length – розмір даного пакету,

address – Internet-адреса відправки,

port – порт комп'ютера, на який відправляється дейтаграмма.

Таблиця 3.5 – Методи класу DatagramPacket

Метод	Опис
`getData()`	Повертає масив байт, що представляють собою вміст дейтаграми
`setData()`	Записує в пакет дані, отримуючи в якості параметрів байтові масиви
`getAddress()` і `setAddress()`	Повертає/встановлює адресу підключення
`getPort()` і `setPort()`	Повертає/встановлює порт підключення
`getLength()` і `setLength()`	Повертає/встановлює розмір дейтаграми

У процесі роботи з об'єктами дейтаграм можуть виникати такі виняткові ситуації:

SocketException – помилка протоколу при зверненні до сокета;

SecurityException – помилка доступу до системи безпеки;

UnknownHostException – невірно вказано адресу InetAddress.

Контрольні питання до розділу 3

1. Для чого призначений і яке значення повертає метод getAllByName() класу java.net.InetAddress?
2. Поясните, в чому полягає відмінність між методами getByName(null) і getLocalHost()?
3. Як отримати вміст сторінки, використовуючи її URL?
4. Які дії необхідно зробити для встановлення TCP з'єднання між двома java-додатками?
6. Які дії необхідно зробити для обміну даними по UDP протоколу?

4 ТЕХНОЛОГІЇ РОЗРОБКИ WEB-ЗАСТОСУВАНЬ

4.1 Розширювана мова розмітки XML

Багато аспектів створення і роботи Web-застосувань пов'язані з обміном різноманітно структурованими даними між окремими компонентами або поданням інформації про організацію, властивості і конфігурації системи, що має гнучку структуризацію. ***Розширювана мови розмітки (Extensible Markup Language, XML)*** є практично універсальним форматом даних. XML – це стандартна лексична форма для представлення текстової інформації різної структури і стандартні ж способи опису цієї структури. Всі сучасні технології розробки Web-застосувань так чи інакше використовують XML.

Надамо короткий огляд основних конструкцій XML. Різні елементи даних в рамках XML-документів виділяються *тегами* – кожен елемент починається з відкриваючого тега `<tag>` і закінчується закриваючим `</tag>`. Тут `tag` – ідентифікатор тега, який зазвичай є англійським словом або набором слів, що розділяються знаками '-', яке (-і) описують призначення цього елемента даних. Елементи даних можуть бути вкладені один в інший, утворюючи дерево документа.

Крім того, кожен елемент може мати набір значень атрибутів, які представляють собою рядки, числа або логічні значення. Значення атрибутів для даного елемента поміщаються всередині його відкриваючого тега. Елемент даних, що не має вкладених піделементи, може бути оформлений у вигляді конструкції `<tag ... />`, тобто не мати окремого закриваючого тега.

Нижче наведено приклад опису інформації про книгу у вигляді XML.

```
<book
    title = "Pattern-Oriented Software Architecture,
             Volume 1: A System of Patterns"
    ISBN = "047195869"
    year = 1 996
    hardcover = true
    pages = 476
    language = "English">
  <author> Frank Buschmann </ author>
  <author> Regine Meunier </ author>
  <author> Hans Rohnert </ author>
```

```
  <author> Peter Sommerlad </ author>
  <author> Michael Stal </ author>
  <publisher
      title = "John Wiley & Sons"
      address = "605 Third Avenue, New York, NY 10158-
                0012, USA" />
</ book>
```

У цьому прикладі тег <book>, що представляє опис книги, має вкладені теги <author> і <publisher>, що представляють її авторів (таких тегів може бути декілька) і видавця. Він також має атрибути title, ISBN, year, hardcover, pages і language (назва книги, її міжнародний стандартний номер, тобто International Standard Book Number або ISBN, плюс рік видання, наявність твердої обкладинки, число сторінок і мова). Тег <publisher>, у свою чергу, має атрибути title і address (назва та юридична адреса видавничої організації).

Розширюваним XML названий тому, що можна задати спеціальну структуру тегів і їх атрибутів для деякого виду документів. Ця структура описується в окремому документі, званому *схемою*, який сам написаний на спеціальному підмножині XML, *DTD (Document Type Declaration, декларація типу документа)* або *XMLSchema*.

XML-документ завжди починається заголовком, що описує версію XML, якій відповідає документ, і використовуване кодування. За замовчуванням використовується кодування UTF-8.

Потім найчастіше йде опис типу документа, що вказує схему, якою він відповідає, і тег кореневого елемента, що містить всі інші елементи даного документа. Схема може задаватися у форматі DTD або XMLSchema. Другий, хоча і є новішим, поки ще використовується рідше, тому що досить багато документів визначається за допомогою DTD і дуже багато інструментів для обробки XML можуть користуватися цим форматом. Використовувана схема визначається відразу двома способами – за допомогою рядка, який може служити ключем для пошуку схеми на даній машині, і за допомогою уніфікованого ідентифікатора документа (Unified Resource Identifier, URI), що містить її опис і використовується в тому випадку, якщо її не вдалося знайти локально.

Нижче наводиться приклад заголовка і опису типу документа для дескриптора розгортання EJB компонентів.

```
<? xml version = "1.0" encoding = "UTF-8"?>
<! DOCTYPE sun-ejb-jar PUBLIC "- // Sun Microsystems, Inc
```

```
.// DTD Application Server 8.1 EJB 2.1 // EN ""
http://www.sun.com/software/appserver/dtds/sun-ejb-jar_2_1-
1.dtd ">
<sun-ejb-jar>
...
</ sun-ejb-jar>
```

Інший приклад показує заголовок документа DocBook, заснованого на XML формату для технічної документації, яка може бути автоматично перетворена в HTML, PDF і інші документи з певними для них правилами верстки.

```
<? xml version = "1.0" encoding = "windows-1251"?>
<! DOCTYPE article PUBLIC "- // OASIS // DTD DocBook XML
V4.3 // EN"
"http://www.oasis-open.org/docbook/xml/4.3/docbookx.dtd">
<article>
...
</ article>
```

Крім елементів даних і заголовка з описом типу документа, XML-документ може містити коментарі, що поміщаються в теги `<!-- ... -->`, інструкції обробки виду `<? processor-name ...?>` (тут `processor-name` – ідентифікатор обробника, якому призначена інструкція) та секції символьних даних CDATA, які починаються набором символів `<! [CDATA [`, а закінчуються за допомогою`]]>`. Усередині секцій символьних даних можуть бути будь-які символи, за винятком закриваючої комбінації. В інших місцях деякі спеціальні символи повинні бути представлені комбінаціями символів відповідно до табл.4.1.

Таблиця 4.1 – Спеціальні символи в XML

Символ	Подання в XML
<	&lt;
>	&gt;
&	&amp;
"	&quot;
'	&apos;

4.2 Платформа Java Enterprise Edition

Платформа Java EE призначена в першу чергу для розробки розподілених Web-застосувань і підтримує наступні 4 види компонентів.

1) ***Enterprise JavaBeans (EJB).*** Компоненти EJB призначені для реалізації на їх основі бізнес-логіки застосування і операцій над даними. Будь-які компоненти, розроблені на Java, прийнято називати *бінами* (bean, боб або квасолина, в розмовній мові має також значення голови і монети). Компоненти Enterprise JavaBean відрізняються від «звичайних» тим, що працюють в рамках EJB-контейнера, який є для них компонентним середовищем. Він підтримує наступні базові служби при роботі з компонентами EJB.

– Автоматичну підтримку звернень до компонентів, розміщених на різних машинах.

– Автоматичну підтримку транзакцій.

– Автоматичну синхронізацію стану баз даних і відповідних компонентів EJB в обидві сторони.

– Автоматичну підтримку захищеності за рахунок аутентифікації користувачів, перевірки прав користувачів або компонентів на виконання виконуваних ними операцій та авторизації вироблених дій.

– Автоматичне управління життєвим циклом компонента (послідовністю переходів між станами типу «відсутній» – «ініціалізований» – «активний») і набором компонентів як ресурсами: видалення компонентів, що стали непотрібними; завантаження нових компонентів; балансування навантаження між наявними компонентами; використання пулу готових до роботи, але не ініціалізованих компонентів, щоб не витрачати час на їх видалення і створення, і ін.

В цілому EJB-контейнер являє собою приклад ***об'єктного монітора транзакцій (object transaction monitor)*** – ПЗ проміжного рівня, що підтримує в рамках об'єктно-орієнтованої парадигми віддалені виклики методів та розподілені транзакції.

2) ***Web-компоненти (Web components).*** Ці компоненти служать для надання інтерфейсу до корпоративних програмних систем поверх широко використовуваних протоколів Інтернет, а саме, HTTP. Надані інтерфейси можуть бути як інтерфейсами для людей (WebUI), так і спеціалізованими програмними інтерфейсами, працюючими подібно віддаленого виклика методів, але поверх HTTP.

До групи Web-компонентів входять *фільтри (filters)*, *обробники Web-подій (web event listeners)*, ***сервлети (servlets)*** і ***серверні сторінки Java (JavaServer Pages, JSP)***.

Компонентної середовищем для роботи Web-компонентів служить ***Web-контейнер***, що поставляється в рамках будь-якої реалізації платформи J2EE. Web-контейнер реалізує такі служби, як управління життєвим циклом компонентів і набором компонентів як ресурсом, розпаралелювання незалежних робіт, виконання віддалених звернень до компонентів, підтримка захищеності за допомогою перевірки прав компонентів і користувачів на виконання різних операцій.

1) *Звичайні застосування на Java.* J2EE є розширенням J2SE і тому всі Java застосування можуть працювати і в цьому середовищі. Однак, на додаток до звичайних можливостей J2SE, ці застосування можуть використовувати у своїй роботі Web-компоненти і EJB, як безпосередньо, так і віддалено, зв'язуючись з ними по HTTP.

2) *Аплети (applets).* Це невеликі компоненти, що мають графічний інтерфейс користувача і призначені для роботи всередині стандартного Web-браузера. Вони використовуються в тих випадках, коли не вистачає виразних можливостей користувальницького інтерфейсу на базі HTML, і можуть зв'язуватися з видаленими Web-компонентами, що працюють на сервері, по HTTP.

Компонент будь-якого з цих видів оформляється як невеликий набір класів і інтерфейсів на Java, а також має *дескриптор розгортання (deployment descriptor)* – опис в певному форматі на основі XML конфігурації компонента в рамках контейнера, в який він поміщається. Застосування в цілому також має дескриптор розгортання. Дескриптори розгортання відіграють важливу роль, дозволяючи міняти деякі параметри функціонування компонента і прив'язувати їх до параметрів середовища, в рамках якого компонент працює, не зачіпаючи його код.

Платформа J2EE пристосована для розробки багаторівневих Web-застосувань. При роботі з такими застосуваннями користувач формує свої запити, заповнюючи HTML-форми в браузері, який упаковує їх в HTTP-повідомлення і пересилає Web-серверу. Web-сервер передає ці повідомлення Web-компонентам, виділяє з них вихідні запити користувача і передає їх для обробки компонентам EJB. Результати роботи EJB компонентів перетворюються Web-компонентами в HTML-сторінки, що динамічно генеруються, і відправляються назад користувачеві, постаючи перед ним у

вікні браузера. Аплети використовуються там, де потрібен більш функціональний інтерфейс, ніж стандартні форми і сторінки HTML.

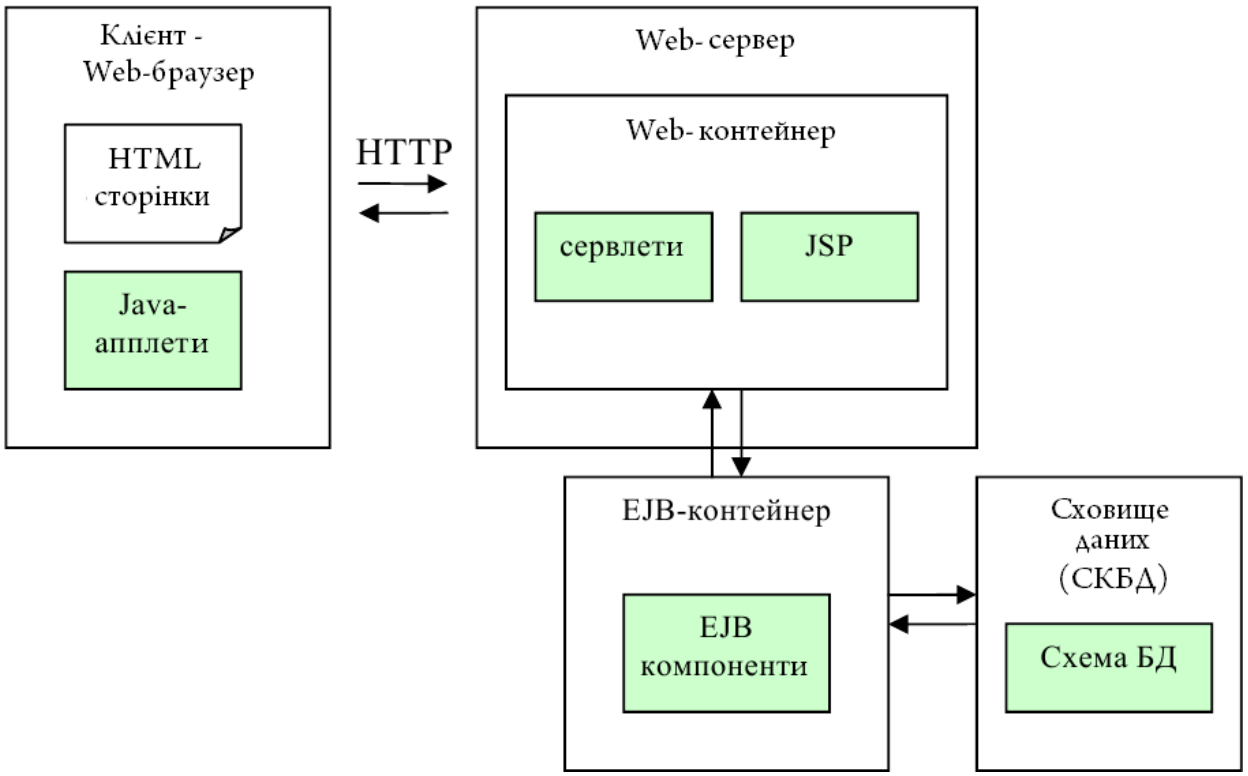

Рисунок 4.1 – Типова архітектура J2EE застосування
Виділено компоненти, що розробляються вручну

Таким чином, застосування на базі J2EE будуються з використанням трьох основних архітектурних стилів.

1) Багаторівнева система. Найбільші підсистеми організовані як рівні, що вирішують різні завдання.

– Інтерфейс взаємодії із зовнішнім середовищем, включаючи користувачів, реалізується за допомогою Web-компонентів.

– Рівень бізнес-логіки і моделі даних реалізується за допомогою EJB компонентів.

– Рівень управління ресурсами будується на основі комерційних систем управління базами даних (СКБД). Можна також підключати інші види ресурсів, для яких мається реалізація інтерфейсу постачальника служб J2EE (J2EE service provider interface, J2EE SPI).

2) Незалежні компоненти. Перші два рівня побудовані з окремих компонентів, кожен з яких має власну область відповідальності, але може

залучати для вирішення приватних завдань інші компоненти.

3) Дані-вид-обробка (MVC). Робота компонентів в рамках обробки групи тісно пов'язаних запитів організовується за зразком MVC. При цьому сервлети і обробники Web-подій служать обработчиками, компоненти JSP – видом, а компоненти EJB – моделлю даних.

Зв'язок між компонентами, що працюють в різних процесах і на різних машинах, забезпечується в J2EE, в основному, двома способами: синхронний зв'язок – за допомогою реалізації віддаленого виклику методів на Java (Java RMI), асинхронна – за допомогою служби повідомлень Java (Java message service, JMS).

Java RMI є прикладом реалізації загальної техніки віддаленого виклику методів. Базові інтерфейси для реалізації віддаленого виклику методів на Java знаходяться в пакеті `java.rmi` стандартної бібліотеки.

Набір методів деякого класу, доступних для віддалених викликів, виділяється в спеціальний інтерфейс, який називається *віддаленим інтерфейсом (remote interface)* і успадковує `java.rmi.Remote`. Сам клас, методи об'єктів якого можна викликати віддалено, повинен реалізовувати цей інтерфейс. Цей же інтерфейс реалізується автоматично створюваною клієнтською заглушкою. Тому об'єкти-клієнти працюють тільки з об'єктом цього інтерфейсу, а не з об'єктом класу, що реалізовує декларовані в такому інтерфейсі операції.

Крім того, клас, який реалізує віддалений інтерфейс, повинен наслідувати класу `java.rmi.server.RemoteObject`. Цей клас містить реалізації методів `hashCode()`, `equals()` і `toString()`, які враховують можливість розміщення його об'єктів у процесах, відмінних від того, де вони викликаються. Зазвичай успадковується не саме цей клас, а його підкласи `java.rmi.server.UnicastRemoteObject` або `java.rmi.activation.Activatable`. Перший реалізує загальну функціональність об'єктів, які можна викликати віддалено поверх транспортного протоколу TCP, поки працює процес, що містить їх, включаючи занесення інформації про такі об'єкти до реєстру RMI (власна служба іменування в рамках Java RMI). Другий клас служить для реалізації *активізуємих об'єктів (activatable objects)*, які створюються системою «на вимогу» – як тільки хтось викликає в них який-небудь метод. Посилання на такі об'єкти можуть зберігатися, а звернутися до них можна через довгий час, навіть після перезавантаження системи.

Кожен ***віддалений метод (remote method)***, тобто метод, який можна

викликати з іншого процесу, повинен декларувати в якості класу можливих винятків `java.rmi.RemoteException` або його базові типи `java.io.IOException або java.lang.Exception`. Цей клас сам слугує базовим для класів винятків, що представляють помилки зв'язку, помилки маршалинга параметрів або результатів і помилки протоколів, що реалізують RMI.

Об'єкти, що реалізують один з віддалених інтерфейсів, можуть бути передані в якості параметрів або результатів віддалених методів за посиланням як об'єкти цього інтерфейсу. При цьому в інший процес передається байт-код клієнтської заглушки, пов'язаної з таким об'єктом, і об'єкти цього процесу отримують можливість викликати в ньому методи.

Решта об'єктів передаються як параметри або результати віддалених викликів за допомогою серіалізації їх даних і побудови копії такого об'єкта в іншому процесі. Так само передаються і створювані віддаленим методом виключення. Для цього їм необхідно реалізовувати інтерфейс `java.io.Serializable` або ж бути значеннями примітивних типів.

Простий приклад Java класів, взаємодіючих по RMI, наведено нижче.

Код віддаленого інтерфейсу.

```
package examples.rmi;

public interface Hello extends java.rmi.Remote
{
  public String hello() throws
                        java.rmi.RemoteException;
}
```

Код реализации удаленного интерфейса.

```
package examples.rmi;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends UnicastRemoteObject
implements Hello
{
  public static final String ServerHost = "hostname";
  public static final String ServerURL = "rmi://" +
ServerHost + ":2001/SERVER";
  public static final int RegistryPort = 2000;
```

```
  public HelloImpl () throws java.rmi.RemoteException { }

  public String hello () throws java.rmi.RemoteException
  { return "Hello!"; }

  public static void main (String[] args)
  {
    try
    {
      Hello stub = new HelloImpl();
      Registry registry =
                 LocateRegistry.getRegistry(RegistryPort);
      registry.rebind(ServerURL, stub);
    }
    catch (Exception e)
    {
      System.out.println("server creation exception");
      e.printStackTrace();
    }
  }
}
```

Код класса-клиента.

```
package examples.rmi;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class HelloClient
{
  public HelloClient () { }

  public static void main (String[] args)
  {
    try
    {
      Registry registry = LocateRegistry.getRegistry
      (HelloImpl.ServerHost, HelloImpl.RegistryPort);
      Hello stub = (Hello)
                 registry.lookup(HelloImpl.ServerURL);
      System.out.println("response: " + stub.hello());
    }
    catch (Exception e)
```

```
        {
          System.err.println("Client exception: " +
                                              e.toString());
          e.printStackTrace();
        }
      }
    }
```

Для того щоб запустити цей приклад, потрібно виконати наступні дії.

1. Скомпілювати всі класи і інтерфейси. У класі-реалізації серверного компонента константа `ServerHost` повинна бути инициализирована ім'ям машини, на якій буде працювати сервер.

2. Створити та скомпілювати заглушки за допомогою компілятора Java RMI, запустивши його в кореневій директорії для коду на Java.

```
rmic examples.rmi.HelloImpl
```

1. Запустити реєстр RMI на тій машині, на якій буде виконуватися серверний компонент `HelloImpl`. Реєстр використовується для реєстрації серверного об'єкта і подальшого пошуку його клієнтами і в даному прикладі приймає повідомлення по порту 2000.

```
start rmiregistry 2000 &
```

2. Запустити сам серверний компонент.

```
 java examples.rmi.HelloImpl
```

5. Запустити клієнтський компонент на тій же машині або на інший.

```
java examples.rmi.HelloClient
```

Якщо ніяких помилок не зроблено, і порти 2000 і 2001 на серверній машині вільні, клієнт видасть повідомлення.

```
response: Hello!
```

Оскільки базові інтерфейси компонентів EJB успадковують java.rmi.Remote, такі компоненти автоматично надають можливість звертатися до деяких зі своїх методів віддалено.

Служба повідомлень Java (JMS) являє собою специфікацію інтерфейсів для підтримки передачі повідомлень (зокрема, асинхронних) між компонентами в рамках платформи J2EE. Її базові бібліотеки – пакет `javax.jms` – не входять до складу J2SE. Та або інша реалізація JMS повинна входити в будь-яку реалізацію платформи J2EE.

Основні елементи JMS наступні.

– *Повідомлення.* Всі об'єкти-повідомлення реалізують інтерфейс `javax.jms.Message`. Повідомлення має тіло, яке може бути відображенням (map) ключів в значення,текстом, об'єктом, набором байт або

потоком значень примітивних типів Java. Кожен з видів повідомлень представлений особливим підінтерфейсом загального інтерфейсу `Message`. Повідомлення може мати набір заголовків (headers), більшість з яких визначаються автоматично. Крім того, повідомлення може мати набір властивостей, які дозволяють визначати додаткові заголовки, специфічні для цього застосування.

– Клієнт може передати повідомлення, встановивши з'єднання з провайдером JMS. З'єднання представляються за допомогою об'єктів інтерфейсу `javax.jms.Connection`, а отримати з'єднання можна за допомогою фабрики з'єднань (об'єкт `javax.jms.ConnectionFactory`), знайшовши її за допомогою служби іменування. Передати повідомлення можна і скориставшись об'єктом-адресатом (об'єкт `javax.jms.Destination`), який також можна отримати через службу іменування.

– JMS підтримує як зв'язок *точка-точка (peer-to-peer, P2P)*, так і посилку і прийом повідомлень за схемою *підписки/публікації*. Основні інтерфейси JMS (з'єднання, фабрика з'єднань, адресат, сесія і ін.) мають специфічні підінтерфейси для кожної з цих двох моделей зв'язку. У клієнтських програмах рекомендується завжди використовувати загальні інтерфейси.

4.3 Розвиток технологій Java EE

В цей час програмисти використовують нові компонентні технології, які націлені на підвищення гнучкості Web-застосувань, зручності їх створення та підтримки, а також на зниження трудомісткості внесення змін у застосування такого роду.

Для вирішення проблем зручності розробки і підтримки додатків J2EE використовуються різні бібліотеки, інструменти та компонентні середовища, створені в співтоваристві Java-розробників.

Java Server Faces. Java Server Faces (JSF) включають бібліотеку елементів управління WebUI `javax.faces` і дві бібліотеки користувальницьких тегів, призначених для використання цих елементів управління в рамках серверних сторінок Java. С допомогою тегів бібліотеки `jsf/html` елементи управління розміщуються на сторінці, а за допомогою тегів з `jsf/core` описується обробка подій, пов'язаних з цими елементами, і

перевірка коректності дій користувача.

В аспекті побудови WebUI на основі серверних сторінок Java технологія Java Server Faces є розвитком підходу Struts (Struts включають рішення і для інших аспектів розробки додатків), пропонуючи більш багаті бібліотеки елементів WebUI і більш гнучку модель управління ними.

На додаток до бібліотек елементів WebUI JSF пропонує визначати правила навігації між сторінками в конфігураційному файлі програми. Кожне правило відноситься до деякої множині сторінок і при виконанні певної дії або виконанні події наказує переходити на деяку сторінку. Дії та події зв'язуються з діями користувача або логічними результатами їх обробки (такими результатами можуть бути, наприклад, успішна реєстрація замовлення в системі, спроба входу користувача в систему з неправильним паролем та ін.).

Управління даними додатка. Hibernate. Технології забезпечення синхронізації внутрішніх даних застосування і його бази даних розвиваються зараз досить активно. Технологія EJB надає відповідні механізми, але за рахунок значного зниження зручності розробки і модифікації компонентів. Забезпечення тієї ж функціональності при більш простій внутрішній організації коду є основним напрямком розвитку в даній області.

Можливим вирішенням цієї задачі є ***об'єктно-реляційні перетворювачі (object-relation mappers, ORM)***, які забезпечують автоматичну синхронізацію між даними застосування у вигляді наборів пов'язаних об'єктів і даними, що зберігаються в системі управління базами даних (СКБД) в реляційному вигляді, тобто у формі записів у кількох таблицях, що посилаються один на одного за допомогою зовнішніх ключів.

Одним з найбільш широко застосовуваних і розвинених в технологічному плані об'єктно реляційних перетворювачів є Hibernate.

Базова парадигма, що лежить в основі обраного Hibernate підходу, – це використання об'єктів звичайних класів Java (оформлених відповідно до вимог специфікації JavaBeans – з чітко виділеними властивостями) в якості об'єктного представлення даних програми. Такий підхід навіть має назву – акронім POJO (plain old Java objects, прості старі Java-об'єкти), покликане показати його відмінність від складних технік побудови компонентів, схожих на EJB.

Велике достоїнство подібного підходу – можливість використовувати один раз створені набори класів, що представляють поняття предметної області, в якості моделі даних будь-яких додатків на основі Java, незалежно

від того, чи є вони розподіленими або локальними, чи потрібне в них синхронізація з базою даних і збереження даних об'єктів чи ні.

Hibernate підтримує зручні засоби для опису складних відповідностей між об'єктами і записами таблиць при використанні успадкування. Так, легко можуть бути підтримані: відображення даних всіх об'єктів класів-спадкоємців в одну таблицю, відображення об'єктів різних класів-спадкоємців в різні таблиці, відображення даних полів загального класу-предка в одну таблицю, а даних класів-спадкоємців – в різні таблиці, а також змішані стратегії подібних відображень.

У застосування на основі Hibernate об'єкти одного і того ж класу можуть бути як збереженими, тобто представляють дані, що зберігаються в базі даних, так і тимчасовими, що не мають відповідних записів у базі даних. Переклад об'єкта з одного з цих видів в інший здійснюється за допомогою всього лише одного виклику методу допоміжного класу середовища Hibernate.

Java Data Objects. Ще більш спростити розробку об'єктно-орієнтованих застосувань, дані яких можуть зберігатися в базах даних, покликана технологія Java Data Objects (JDO).

В її основі теж лежить використання для роботи з збереженими даними звичайних класів на Java, але в якості сховища даних може виступати не тільки реляційна СКБД, але взагалі будь-яке сховище даних, що має відповідний спеціалізований адаптер (в рамках JDO це має бути реалізація інтерфейсу `javax.jdo.PersistenceManager`). Основна функція цього адаптера – прозора для розробників синхронізація сховища даних і набору збережених об'єктів в пам'яті програми. Він повинен також забезпечувати досить високу продуктивність програми, незважаючи на наявність декількох проміжних шарів між класами самого застосування і сховищем даних, що представляються ними.

Використання JDO дуже схоже на використання Hibernate. Конфігураційні файли, зберігають інформацію про прив'язку об'єктів Java класів до записів у певних таблицях, а також про прив'язку колекцій посилань на об'єкти до посиланнями між записами, також схожі на аналогічні файли Hibernate. Зазвичай перші навіть дещо простіше, оскільки більшу частину роботи по відображенню полів об'єктів в поля записів бази даних бере на себе спеціалізований адаптер.

Також JDO надає засоби для побудови запитів за допомогою опису властивостей об'єктів, без використання більш звичного вбудованого SQL.

В цілому, підхід JDO є узагальненням підходу ORM на довільні сховища даних, але він вимагає реалізації більш складних спеціалізованих адаптерів для кожного виду таких сховищ, в той час як один і той же ORM може використовуватися для різних реляційних СКБД, вимагаючи для своєї роботи тільки наявності більш простого драйвера JDBC.

Середовище Spring. Середовище Spring – одне з найбільш технологічних на даний момент середовищ розробки Web-застосувань. В ньому отримали подальший розвиток ідеї спеціалізації обробників запитів, використані в Struts. У Spring також використовуються елементи аспектно-орієнтованого підходу до розробки ПЗ, що дозволяють значно підвищити гнучкість і зручність модифікації побудованих на його основі Web-застосувань.

Основне завдання, на вирішення якої націлене середовище Spring, – інтеграція різнорідних механізмів, використовуваних при розробці компонентних Web-застосувань. В якості двох основних засобів такої інтеграції використовуються ідеї *інверсії управління (inversion of control)* і *аспектно-орієнтованого програмування (aspect-oriented programming, AOP).*

Інверсією управління називають відсутність звернень з коду компонентів застосування до якого-небудь API, що надається середовищем і її бібліотеками. Замість цього компоненти застосування реалізовують тільки функції, необхідні для роботи в рамках предметної області і вирішення тих завдань, з якими застосуванню доведеться мати справу. Побудова з цих компонентів готового застосування, конфігурація окремих його елементів і зв'язків між ними – це справа середовища, яка сама в певні моменти звертається до потрібних операцій компонентів. Конфігурація компонентів застосування в середовищі Spring здійснюється за допомогою XML-файла springapp-servlet.xml. Цей файл містить опис набору компонентів, які налаштовуються за допомогою вказівки параметрів ініціалізації відповідних класів і значень своїх властивостей в сенсі JavaBeans.

Інша назва механізму інфверсії управління – *вбудовування залежностей* (dependency injection) – пов'язано з можливістю вказувати в коді програми лише інтерфейси деяких об'єктів, а їх точний клас, як і спосіб їх отримання (створення нового об'єкта, пошук за допомогою служби каталогів, звернення до фабриці об'єктів та ін.) – описувати тільки в конфігураційному файлі. Такий механізм дозволяє розділити використання об'єктів і їх конфігурацію і міняти їх незалежно один від одного.

Аналогічний механізм використовується і в EJB 3.0 в наступному

вигляді. За допомогою конструкції `Resource Type object;` деякий об'єкт може бути оголошений у коді як такий, що підлягає окремій конфігурації, а в конфігураційному файлі для об'єкта з ім'ям `object` вказується його точний тип і спосіб ініціалізації.

Аспектно-орієнтований підхід до програмування грунтуються на виділенні аспектів – окремих завдань, що вирішуються застосуванням – таким чином, щоб їх рішення можна було організувати у вигляді виконання певних дій кожного разу, коли виконується певний елемент коду в ході роботи програми. При цьому дії в рамках даного аспекту не повинні залежати від інших аспектів та інших дій, виконуваних програмою. Не всі завдання можуть бути представлені як аспекти. Тому AOP-програма являє собою композицію зі «звичайної» програми, що описується поза рамками AOP, і аспектних дій, що виконуються в певних точках «звичайної» програми. Такі дії називають *вказівками (advice)*, точки їх виконання в «звичайній» програмі – *точками вставки (joinpoints)*, а набори точок вставки, в яких повинна виконуватися одна і та ж дія – *перетинами (pointcuts)*.

Прикладом вказівок можуть служити трасування параметрів виклику методу або перевірка коректності ініціалізації полів об'єкта. В якості прикладів точок вставки можна навести момент перед викликом певного методу або відразу після такого виклику, момент після ініціалізації певного об'єкта або перед його знищенням. Перетини можуть описуватися умовами типу «перед викликом в даному об'єкті методу, чиє ім'я починається на"get"» або «Перед знищенням об'єкта класу A зі значенням поля value, що перевищує 0».

Spring дає можливість визначати перетини і вказівки, що виконуються в них, в конфігураційних файлах програми.

За допомогою AOP в Spring реалізована підтримка інтеграції додатків з сховищами даних за допомогою різних технологій, прикладами яких є JDO і Hibernate. Для того щоб використовувати будь-яку таку технологію в застосуванні на основі Spring, достатньо мати спеціалізований адаптер, який реалізує інтерфейс, пропонований Spring для підтримки синхронізації даних між застосуванням і сховищем даних. Після вказівки цього адаптера в конфігурації програми середовище саме подбає про те, щоб кожного разу після появи можливих відмінностей між даними застосування і бази даних були викликані методи цього адаптера, що копіюють нові дані в ту чи іншу сторону.

Схожим чином підтримується інтеграція з різними реалізаціями служб підтримки транзакцій і декларативне управління транзакціями. Методам звичайного класу Java в конфігураційному файлі (або за допомогою анотацій Java 5) можна приписати певні транзакційні атрибути, а також набір типів виняткових ситуацій, що викликають відкат транзакції. Для порівняння – в EJB 2.1 тільки виключення, чий тип є спадкоємцем `java.lang.RuntimeException`, `java.lang.Error` або `javax.ejb.EJBException`, викликають автоматичний відкат транзакції. Адаптер конкретної реалізації служби транзакцій також вказується в конфігурації програми.

Використання інверсії управління дозволяє також спростити опис конфігурації сервлетів і *контролерів* (Spring-аналог дій з Struts) і визначення самих контролерів.

Ajax. Розповідаючи про розвиток технологій розробки Web-застосунків, неможливо оминути увагою набір технік, відомий під назвою Ajax і використовуваний для зниження часу реакції Web-інтерфейсів на дії користувача.

Власне кажучи, Web-технології не дуже добре пристосовані для побудови користувальницького інтерфейсу інтерактивних застосунків, тобто таких, де користувач досить часто виконує якісь дії, на які застосунок повинен реагувати. Вони спочатку розроблялися для надання доступу до статичної інформації, яка змінюється рідко і представлена у вигляді набору HTML-сторінок. Зазвичай при обміні даними між Web-клієнтом і Web-сервером клієнт зрідка посилає серверу прості і невеликі за обсягом запити, а той у відповідь може надсилати досить об'ємні документи.

У інтерактивних застосунках обмін даними між інтерфейсними елементами застосунка та обработчиками запитів дещо інший. Оброблювач досить часто отримує запити і невеликі набори їх параметрів, а зміни, які відбуваються в інтерфейсі після отримання відповіді на запит, зазвичай теж невеликі. Часто потрібно змінити зміст лише частини показуваної браузером сторінки, в той час як інші її елементи представляють більш стабільну інформацію, що є елементом дизайну сайту або набором пунктів його меню. Для відображення цих змін не обов'язково пересилати з сервера весь HTML-документ, що передбачається в рамках традиційного обміну інформацією за допомогою Web. Точно так само, якби коректність даних, що вводяться користувачем, можна було б перевірити на стороні клієнта, обробка некоректного введення відбувалася б набагато швидше і не вимагала б

взагалі ніякого обміну даними з сервером.

Ajax намагається вирішити ці завдання за допомогою комбінації коду на JavaScript, що виконується в браузері, і спеціальних XML- повідомлень, що передаються час від часу між клієнтом і сервером, і містять тільки істотну інформацію про запит або зміни HTML- сторінки, які повинні бути показані. В рамках браузера в окремому потоці працює ядро Ajax, яке отримує повідомлення JavaScript-коду про виконання користувачем певних дій, виконує перевірку їх коректності, перетворює їх в посилку відповідного запиту на сервер, перетворює відповідь сервера в нову сторінку або ж видає вже наявну в спеціальному кеші відповідь. Запити на сервер і їх обробка здійснюються часто асинхронно з діями користувачами, дозволяючи помітно знизити час реакції системи на них.

Навчальне електронне видання

КУЗНІЧЕНКО СВІТЛАНА ДМИТРІВНА

КРОС-ПЛАТФОРМНЕ ПРОГРАМУВАННЯ

Конспект лекцій

Видавець і виготовлювач
Одеський державний екологічний університет
вул. Львівська, 15, м. Одеса, 65016
тел./факс: (0482) 32-67-35
E-mail: info@odeku.edu.ua
Свідоцтво суб'єкта видавничої справи
ДК № 5242 від 08.11.2016