А.В. Гаврилов, С.В. Клименков, Е.А. Цопа

Программирование на Java

Конспект лекций

Санкт-Петербург
2010

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

А.В. Гаврилов, С.В. Клименков, Е.А. Цопа

Программирование на Java

Конспект лекций

Санкт-Петербург

2010

Гаврилов А.В., Клименков С.В., Цопа Е.А.
Программирование на Java. Конспект лекций – СПб: СПбГУ ИТМО, 2010. – 130 с.

Данное пособие представляет собой краткий справочник по языку Java и может использоваться как конспект лекционного курса «Системы программирования Интернет-приложений». Рассмотрены основные концепции объектно-ориентированного программирования, описан синтаксис языка Java, а также приведено описание основных классов и интерфейсов, входящих в стандартную библиотеку с алгоритмами и примерами их использования.

Для подготовки бакалавров и магистров по направлению 23.01.00 «Информатика и вычислительная техника»; по программам подготовки магистров 23.01.00.11 «Базы данных» и 23.01.00.13 «Сети ЭВМ и телекоммуникации».

Рекомендовано к печати Ученым советом факультета КТиУ, протокол №15 от 16.11.2010.

В 2009 году Университет стал победителем многоэтапного конкурса, в результате которого определены 12 ведущих университетов России, которым присвоена категория «Национальный исследовательский университет». Министерством образования и науки Российской Федерации была утверждена Программа развития государственного образовательного учреждения высшего профессионального образования «Санкт-Петербургский государственный университет информационных технологий, механики и оптики» на 2009–2018 годы.

Оглавление

Введение. Основные концепции ООП.

Язык программирования Java

- *простой*
  - сходство с C и C++
  - устранение проблематичных элементов
- *объектно-ориентированный*
  - чистая реализация объектно-ориентированной концепции
- *распределенный*
  - поддержка сетевого взаимодействия
  - удаленный вызов методов
- *интерпретируемый*
  - байт-код выполняется виртуальной машиной Java (JVM)
- *надежный*
  - устранение большинства ошибок на этапе компиляции
- *безопасный*
  - контроль и ограничение доступа
- *архитектурно-нейтральный*
  - работа на любых платформах
- *переносимый*
  - независимость спецификации от реализации
- *высокоэффективный*
  - приближенность байт-кода к машинному
  - сочетание производительности и переносимости
- *многопотоковый*
  - встроенная поддержка многопотокового выполнения приложений
- *динамический*
  - загрузка классов во время выполнения приложений

Объекты и классы

Объект - это программная модель объектов реального мира или абстрактных понятий, представляющая собой совокупность *переменных*, задающих состояние объекта, и связанных с ними *методов*, определяющих поведение объекта.

Класс - это прототип, описывающий переменные и методы, определяющие характеристики объектов данного класса.

Характеристики объектов

Класс	***Состояние***	***Поведение***
собака	кличка, возраст, порода, цвет	лает, кусает, виляет хвостом, грызет новые тапочки
автомобиль	цвет, марка, скорость, передача	ускоряется, тормозит, меняет передачу
Point	x, y	show(), hide()
	переменные	*Методы*

Создание объектов

Процесс создания новых объектов по описанию, объявленному в классе, называется *созданием* или *реализацией* (*instantiation*).

Объекты создаются с помощью команды **new**.

Классы и объекты:

Класс	*Объявление переменных*	***Объект***	*Переменные*
Car	color	*myCar*	color = red
	speed		speed = 90
	brand		brand = Volvo
Point	x	*a*	x = 40
	y		y = 25
	isVisible		isVisible = true

Объявление класса	***Реализация объекта***

`class Point {`
`  int x, y;`
`  boolean isVisible;`
`}`
	`Point a; // объявление объекта`
	
	`a = new Point(); // реализация`
	
	`a.x = 40;`
	`a.y = 25;`
	`a.isVisible = true;`

Конструкторы

Для инициализации объекта при его создании используются *конструкторы*.

Объявление класса	***Создание объекта***

`class Point {`
`  int x, y;`
`  boolean isVisible =`
`  false;`
 	
`  Point() {`
`  x = y = 0;`
`  }`
 	
`  Point(x0, y0) {`
`  x = x0;`
`  y = y0;`
`  }`
`}`
	`Point a = new Point();`
	`a.isVisible = true;`
	
	`Point b = new Point(20,50);`

Значения переменных объектов:

Point	***a***	***b***
`int x`	0	20
`int y`	0	50
`boolean isVisible`	true	false

Сообщения

Объекты взамодействуют между собой путем посылки друг другу *сообщений*, которые могут содержать *параметры*. Посылка сообщения осуществляется с помощью вызова соответствующего метода объекта.

Объявление класса	***Вызов методов***

`class Point {`
`  int x, y;`
`  boolean isVisible;`
 	
`void hide() {`
`  isVisible = false;`
	`Point a = new Point(20,30);`
	`a.isVisible = true;`
	`a.hide();`
	
	`a.move(60,80);`
	`a.show();`

```
  }

  void show() {
    isVisible = true;
  }

  void move(x1, y1) {
    x = x1;
    y = y1;
  }
}
```

```
Point center = new Point();
center.show();
```

Компоненты сообщения:

Объект	***Имя метода***	***Параметры***
`a.`	`move`	`(60,80)`
`center.`	`show`	`()`

Инкапсуляция

> *Инкапсуляция* - сокрытие данных внутри объекта, и обеспечение доступа к ним с помощью общедоступных методов

Объявление класса	***Доступ к переменным класса***

`public class Point {`
`  private int x, y;`
 	
`  public void move(x1,y1)`
`  {`
`  x = x1;`
`  y = y1;`
`  }`
 	
`  public int getX() {`
`  return x;`
`  }`
 	
`  public int getY() {`
`  return y;`
`  }`
`}`
	`Point a = new Point(20,30);`
	`int z;`
	
	~~**`a.x = 25`**~~`; // запрещено`
	~~**`z = a.y`**~~`; // запрещено`
	
	`z = a.getY();`
	`a.move(25,a.getY());`
	
	`a.move(a.getX()-5,a.getY()+5);`

Наследование

Наследование или *расширение* – приобретение одним классом (*подклассом*) свойств другого класса (*суперкласса*).

Объявление суперкласса	***Наследование***

`public class Point {`
 	
`  protected int x, y;`
 	
`  public Point() {`
`  x = y = 0;`
`  }`
 	
`  public Point(int x, int y){`
`  this.x = x;`
`  this.y = y;`
`  }`
 	
`  public int getX() {`
`  return x;`
`  }`
 	
`  public int getY() {`
`  return y;`
`  }`
`}`
	`public class Pixel`
	`  extends Point {`
	
	`  private int color;`
	
	`  public Pixel() {`
	`  color = 0;`
	`  }`
	
	`  public Pixel(int x, int y){`
	`  super(x,y);`
	`  color = 0;`
	`  }`
	
	`  public setColor(int c) {`
	`  color = c;`
	`  }`
	
	`  public int getColor() {`
	`  return color;`
	`  }`
	`}`

`Point a = new Point(30,40);`
`int ax, ay;`
`ax = a.getX();`
`ay = a.getY();`
	`Pixel b = new Pixel(20,50);`
	`b.setColor(2);`
	`int bx, by, bc;`
	`bx = b.getX();`
	`by = b.getY();`
	`bc = b.getColor();`

Полиморфизм

Полиморфизм - способность объекта принимать различные формы, что позволяет использовать один и тот же интерфейс для общего класса действий.

Объявление классов	*Использование полиморфизма*

`public class Shape {`
`  ...`
`  abstract void draw();`
`}`
 	
`public class Rectangle`
`  extends Shape {`
`  ...`
`  void draw() {`
`  ...`
`  }`
`}`
 	
`public class Circle`
`  extends Shape {`
`  ...`
`  void draw() {`
`  ...`
`  }`
`}`
	`public class FiguresSet {`
	`  ...`
	`  void drawFigure(Shape s){`
	`  s.draw();`
	`  }`
	
	`  public static void main`
	`  (String args[]) {`
	`  ...`
	`  Rectangle rect;`
	`  Circle circ;`
	`  ...`
	`  rect = new Rectangle();`
	`  circ = new Circle();`
	`  ...`
	`  drawFigure(rect);`
	`  drawFigure(circ);`
	`  }`
	`}`

Принцип подстановки Барбары Лисков

Принцип подстановки Барбары Лисков (Liskov Substitution Principle, LSP) – это важный критерий, используемый при построении иерархии наследования:

> Наследующий класс должен дополнять, а не замещать поведение базового класса.

Таким образом, если объектно-ориентированная модель спроектирована в соответствии с LSP, замена в коде объектов класса-предка на объекты класса-потомка не приведёт к изменениям в работе программы.

Интерфейсы

> *Интерфейс* - абстрактное описание набора методов и констант, необходимых для реализации определенной функции.

Объявление интерфейса	*Воплощение интерфейса*
`interface Displayable {` `  void hide();`	`public class Pixel` `  extends Point`

```
  void show();
}
```

```
  implements Displayable {

  ...

  void hide() {
    ...
  }
  void show() {
    ...
  }
}
```

Класс может воплощать любое количество интерфейсов.

Вложенные, локальные и анонимные классы

Вложенный класс – это класс, объявленный внутри объявления другого класса.

```
public class EnclosingClass {
  ...
  public class NestedClass {
    ...
  }
}
```

Локальный класс – это класс, объявленный внутри блока кода. Область видимости локального класса ограничена тем блоком, внутри которого он объявлен.

```
public class EnclosingClass {
  ...
  public void enclosingMethod(){
    // Этот класс доступен только внутри enclosingMethod()
    public class LocalClass {
      ...
    }
  }
}
```

Анонимный класс – это локальный класс без имени.

```
// Параметр конструктора – экземпляр анонимного класса,
// реализующего интерфейс Runnable
new Thread(new Runnable() {
    public void run() {...}
}).start();
```

Связность

Связность (cohesion) — это мера сфокусированности обязанностей класса. Считается что класс обладает высокой степенью связности, если его обязанности тесно связаны между собой и он не выполняет огромных объемов разнородной работы.

Высокая степень связности (*high cohesion*) позволяет добиться лучшей читаемости кода класса и повысить эффективность его повторного использования.

Класс с *низкой степенью связности* (*low cohesion*) выполняет много разнородных функций или несвязанных между собой обязанностей. Такие классы создавать нежелательно, поскольку они приводят к возникновению следующих проблем:

- Трудность понимания.
- Сложность при повторном использовании.
- Сложность поддержки.
- Ненадежность, постоянная подверженность изменениям.

Классы со слабой связностью, как правило, являются слишком «абстрактными» или выполняют обязанности, которые можно легко распределить между другими объектами.

Связанность

Связанность (coupling) — это мера, определяющая, насколько жестко один элемент связан с другими элементами, либо каким количеством данных о других элементах он обладает.

Элемент с *низкой степенью связанности* (или слабым связыванием, *low coupling*) зависит от не очень большого числа других элементов и имеет следующие свойства:

- Малое число зависимостей между классами.
- Слабая зависимость одного класса от изменений в другом.
- Высокая степень повторного использования классов.

Средства разработки

Инструментальные средства JDK 1.6

javac	компилятор
java	интерпретатор
jre	интерпретатор для конечных пользователей
jdb	отладчик
jvisualvm	профилировщик
javah	генератор файлов заголовков и исходных текстов на C
javap	дизассемблер классов
javadoc	генератор документации
appletviewer	программа просмотра апплетов
jar	программа, позволяющая упаковать множество классов в исполняемый архив
javaws	программа, осуществляющая загрузку и запуск приложений с удалённых web-серверов
jconsole	консоль, предназначенная для мониторинга и управления исполнением приложений
jmap	вывод карты памяти процесса
jps	вывод информации о запущенных процессах
keytool	программа, управляющая ключами и сертификатами
native2ascii	конвертер файлов в ASCII-формат
orbd	ORB-сервер
servertool	консоль администратора ORB-сервера
pack200	архиватор
unpack200	программа для распаковки архивов
policytool	программа для чтения и модификации policy-файлов
rmic	RMI-компилятор
rmid	RMI-сервер
rmiregistry	реестр RMI-объектов
schemagen	генератор XML-схем
serialver	вывод serialVersionUID классов в CLASSPATH
tnameserv	служба имён
wsgen	генератор stubs для веб-сервисов
wsimport	генератор ties для веб-сервисов
xjc	генератор классов на основе XML-схем

Переменная окружения CLASSPATH определяет дополнительные пути поиска классов. Путь по умолчанию указывает на jar-архивы с классами Java API, входящими в состав JDK, которые находятся в каталогах lib и jre/lib. В переменной CLASSPATH через символ : (двоеточие) перечисляются директории, zip- и jar-архивы, содержащие классы, необходимые для выполнения программ.

Для установки переменной CLASSPATH в UNIX используются следующие команды:

1. Bourne или Korn shell:
   `CLASSPATH=.:/usr/local/java/swing/classes`
   export CLASSPATH
2. C-shell:
   `setenv CLASSPATH .:/usr/local/java/swing/classes`

javac

Компилирует исходные тексты (файлы с расширением .java) в байт-код (файлы с расширением .class).

`javac [ `*`параметры`*` ] `*`файлы`*

В одном файле с расширением .java должен находиться только один public-класс, и имя этого класса (без имени пакета) должно совпадать с именем файла (без расширения).

Параметры:

`-classpath `*`путь`*

Переопределяет путь поиска классов, заданный переменной `CLASSPATH`.

`-d `*`каталог`*

Задает каталог для хранения классов (по умолчанию используется текущий каталог). Файлы классов размещаются в подкаталогах в соответствии с именами пакетов классов.

`-deprecation`

Устанавливает режим выдачи сообщения при каждом использовании устаревшего API.

`-nowrite`

Устанавливает режим проверки, при котором откомпилированный класс не записывается.

`-O`

Разрешает использовать оптимизацию классов.

`-verbose`

Устанавливается режим выдачи сообщений о ходе компиляции.

java

Интерпретатор байт-кода. Запускает Java-программы (файлы с расширением .class).

`java [ `*`параметры`*` ] `*`имя_класса`*` [ `*`аргументы`*` ]`

Программа, которую необходимо выполнить, должна представлять собой класс с именем *имя_класса* (без расширения .class, но с указанием пакета, которому принадлежит класс) и содержать метод main() с описанием:

```
public static void main(String args[])
```

Аргументы, указанные в командной строке, помещаются в массив args[] и передаются методу main()

Параметры:

`-classpath `*`путь`*

Переопределяет путь поиска классов, заданный переменной `CLASSPATH`.

`-cs (-checksource)`

Указывают на необходимость сравнения времени модификации файла класса и исходного текста и при необходимости перекомпиляции программы.

`-D`*`имя=значение`*

Присваивает системному свойству с заданным именем указанное значение.

`-noverify`

Отключает режим проверки байт-кода.

`-verify`

Включает режим проверки байт-кода для всех классов (по умолчанию проверяются только классы, загруженные по сети).

`-verbose`

Устанавливается режим выдачи сообщений о загрузке классов.

javadoc

Создает документацию в формате HTML для указанных пакетов или файлов исходных текстов Java.

`javadoc [ `*`параметры`*` ] `*`файлы`*

```
javadoc [ параметры ] пакет
```

Данные для документирования берутся из комментариев для документации, имеющих вид `/**` комментарий `*/`, в которых могут использоваться форматирующие метки HTML.

Параметры:

`-classpath` *`путь`*
: Переопределяет путь поиска классов, заданный переменной `CLASSPATH`.

`-d` *`каталог`*
: Задает каталог для записи документации.

`-docecoding` *`кодировка`*
: Задает кодировку символов для документации.

`-encoding` *`кодировка`*
: Задает кодировку символов для исходных текстов.

`-author`
: Включает в документацию информацию об авторе.

`-version`
: Включает в документацию информацию о версии.

`-verbose`
: Устанавливается режим выдачи сообщений о ходе документирования.

Документирующие комментарии

```
/**
 * Первое предложение является кратким описанием класса или
 * метода. Далее следуют более подробное дополнительное
 * описание. После описаний могут располагаться специальные
 * теги, обеспечивающие дополнительное форматирование.
 * Для классов могут употребляться следующие теги:
 * @author автор
 * @version версия
 * @see класс
 * @see пакет.класс#метод
 * @deprecated объяснение
 */
public class myClass {

    /**
     * Эта переменная содержит очень полезную информацию.
```

```
     * Комментарии должны предшествовать объявлению класса,
     * метода или переменной.
     * Для переменных используются:
     * @see класс
     * @see класс#метод
     * @deprecated объяснение
     */
    public int myVariable;

    /**
     * Метод, устанавливающий все необходимые значения.
     * Для методов, кроме тегов, используемых для переменных,
     * могут также использоваться:
     * @param параметр описание
     * @return описание
     * @exception исключение описание
     */
    public String myMethod(int a, int b) throws myException {
        return str;
    }
}
```

appletviewer

Используется для просмотра апплетов, ссылки на которые имеются в HTML-документах.

`appletviewer [ `*`параметры`*` ] `*`url/файлы`*

Указанные HTML-файлы загружаются, и все апплеты, на которые в них имеются ссылки в виде тега `<APPLET>`, отображаются каждый в собственном окне.

Параметры:

`-debug`
: Апплет запускается под управлением отладчика.

`-`*`Jаргумент`*
: Передает указанный аргумент командной строки интерпретатору.

`-encoding `*`кодировка`*
: Задает кодировку символов для HTML-документов.

Элементы HTML для поддержки Java

```
<APPLET
    CODEBASE = codebaseURL
    ARCHIVE = archiveList
    CODE = appletFile or OBJECT = serializedApplet
    ALT = alternateText
    NAME = appletInstanceName
    WIDTH = pixels  HEIGHT = pixels
    ALIGN = alignment
    VSPACE = pixels  HSPACE = pixels>
    <PARAM NAME = appletAttribute1 VALUE = value>
    <PARAM NAME = appletAttribute2 VALUE = value>
    ...
</APPLET>
```

`<APPLET>`

`CODE`

Указывает имя класса, содержащего апплет. В теге <APPLET> должен присутствовать либо атрибут CODE, либо OBJECT.

`OBJECT`

Указывает имя файла, содержащего сериализованный апплет. При запуске такого апплета вызывается метод `start()`.

`WIDTH,HEIGHT`

Указывают ширину и высоту области для апплета.

`CODEBASE`

Указывает базовый URL или каталог апплета. По умолчанию используется URL HTML-документа.

`ARCHIVE`

Указывает список архивов, загружаемых перед выполнением апплета.

`<PARAM>`

`NAME`

Имя параметра.

`VALUE`

Значение параметра.

Интегрированные среды разработки

- NetBeans (Sun Microsystems / Oracle)
  http://www.netbeans.org

- Eclipse (Eclipse Foundation)
  http://www.eclipse.org
- IntelliJIDEA (JetBrains)
  http://www.jetbrains.com/idea/
- JBuilder (Borland/CodeGear)
  http://www.embarcadero.com/products/jbuilder
- JDeveloper (Oracle)
  http://www.oracle.com/technology/products/jdev/index.html

Синтаксис и структура языка

Приложение Hello, World!

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

```
/**
 * Программа пишет "Hello, World" в стандартный поток вывода
 * @version 2
 */
public class HelloWorld {
  private String name;

  public static void main(String args[]) {
    HelloWorld h;
    if (args.length > 0)
      h = new HelloWorld(args[0]);
    else
      h = new HelloWorld("World");
  }

  /**
   * Конструктор создает объект с заданным именем
   */
  public HelloWorld(String s) {
    name = s;
  }

 /**
  * Метод выводит "Hello" и имя в стандартный поток вывода
  */
  public void sayHello() {
    System.out.println("Hello, " + name + "!");
  }
}
```

1. Создайте текстовый файл с именем HelloWorld.java, содержащий приведенную программу.
2. Скомпилируйте программу:
   `javac HelloWorld.java`

3. Получившийся в результате файл HelloWorld.class запустите на выполнение:
   `java HelloWorld`
4. Создайте документацию (для второго варианта программы):
   `javadoc -version HelloWorld.java`

Апплет Hello, World!

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet {
  public void paint(Graphics g) {
    g.drawString("Hello world!", 50, 25);
  }
}
```

```
<HTML>
<HEAD>
<TITLE> A Simple Program </TITLE>
</HEAD>
<BODY>

Здесь то, что выводит апплет
<APPLET CODE="HelloWorld.class" WIDTH=150 HEIGHT=25>
</APPLET>
</BODY>
</HTML>
```

1. Создайте текстовый файл с именем `HelloWorld.java`, содержащий приведенную программу на языке Java.
2. Скомпилируйте программу:
   `javac HelloWorld.java`
3. Создайте файл `HelloWorld.html`, содержащий программу на языке HTML.
4. Просмотрите получившийся апплет:
   `appletviewer HelloWorld.html`

Класс Applet

Для создания собственного апплета используется подкласс данного класса и переопределяются следующие методы (если необходимо):

`init()`	вызывается при инициализации апплета
`start()`	вызывается при начале просмотра апплета
`stop()`	вызывается при прекращении просмотра апплета
`destroy()`	вызывается при уничтожении апплета

`paint()`	наследуется из `Component` и используется для прорисовки изображения
`getParameter(String name)`	используется для получения параметров, переданных апплету

Безопасность при работе с апплетами

Апплет, загруженный по сети **не может:**

- производить чтение и запись информации в локальной файловой системе;
- создавать сетевые соединения (кроме компьютера, с которого был загружен апплет);
- осуществлять выход, запускать процессы и загружать библиотеки;
- производить печать, использовать системный буфер и системную очередь событий;
- использовать системные свойства (кроме специально оговоренных);
- получать доступ к потокам или группе потоков других приложений;
- пользоваться классом `ClassLoader`;
- пользоваться методами отражения для получения информации о защищенных или частных членах классов;
- работать с системой безопасности.

Типы данных

Тип	*Размер, бит*	*Диапазон значений*
`byte`	8	-128 … 127
`short`	16	-32768 … 32767
`int`	32	-2147483648 … 2147483647
`long`	64	-9223372036854775808 … 9223372036854775807
`float`	32 (IEEE 754)	3.4e-038 … 3.4e+038
`double`	64 (IEEE 754)	1.7e-308 … 1.7e+308
`char`	16	0 … 65536 (символы Unicode)
`boolean`		false, true

```
byte a;
short b = 32;
int c = 1000000;
int cOctal = 011;
itn cHex = 0xCAFE;
long d = 123456789012345L;
```

```
float e = 3.1415926F;
double doublePrecisionValue = 1.23456789E-100;
char symbol = 'a';
char cyrillic = '\u0401';
char newLine = '\n';
boolean toggle = true;
int arr[] = {1,2,3,4,5,6};

final double ¶ = 3.141592653589793238 46;
```

Переменные

Имя *переменной* должно состоять из символов Unicode. Оно не должно совпадать с любым из ключевых слов языка Java, а также с логическими константами true и false. Две переменных не могут иметь одинаковые имена, если они находятся в одной *области видимости*.

Ключевые слова Java

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

Зарезервированные литералы — null, true и false

Операторы

Приоритеты операторов

постфиксные операторы	[] . () expr++ expr--
унарные операторы	++expr --expr +expr -expr ~ !
операции с типами	new (cast)expr
умножение/деление	* / %
сложение/вычитание	+ -
операторы сдвига	<< >> >>>
операторы отношения	< > <= >= instanceof
операторы равенства	== !=

поразрядное И	`&`
поразрядное искл. ИЛИ	`^`
поразрядное ИЛИ	`|`
логическое И	`&&`
логическое ИЛИ	`||`
условный оператор	`?:`
операторы присваивания	`= += -= *= /= %= >>= <<= >>>= &= ^= |=`

Области видимости

Области видимости различных категорий переменных, использующихся в классах и их методах, приведены на рис. 1.

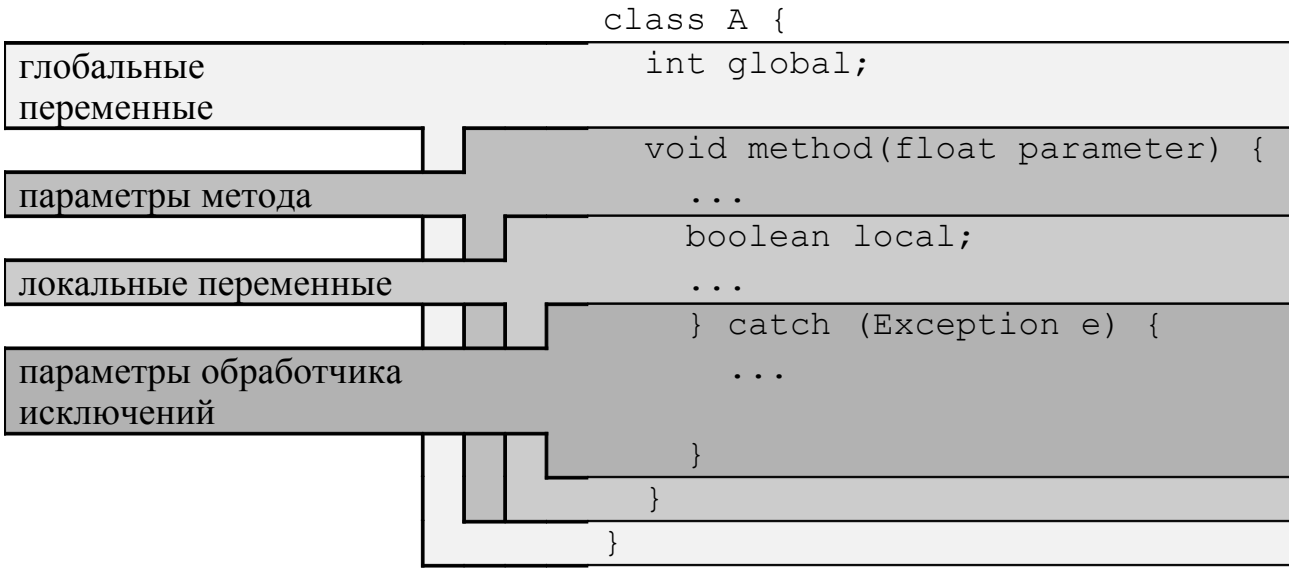

Рисунок 1. Области видимости переменных

Управляющие конструкции

```
if (boolean expr) {
  op1;
} else {
  op2;
}

switch (int expr) {
  case const1:
    op1;
  case const2:
    op2;
    break;
```

```
for (initialization;
     boolean expr;
     increment) {
  op;
}

// array - массив
for (Type x : array) {
  op;
}
```

```
  default:
    op3;
}

// Type — перечисляемый тип
switch (Type x) {
  case CONST_1:
    op1;
    break;
  case CONST_2:
    op2;
    break;
  default:
    op3;
}
```

```
// collection - коллекция
for (Type x : collection){
  op;
}

while (boolean expr) {
  op;
}

do {
  op;
} while (boolean expr);
```

```
try {
  op;
} catch (Exception e) {
  block1;
} finally {
  block2;
}

throw new Exception();
```

```
label:
  for (...) {
    for (...) {
      op;
      continue label;
    }
  }

return;
```

Модификаторы

`abstract`	метод только описан, но не реализован; класс содержит абстрактные методы
`final`	переменная является константой; метод не может быть переопределен; класс не может быть расширен
`static`	переменная является переменной класса; метод является методом класса; инициализатор запускается при загрузке класса; класс является классом верхнего уровня
`public`	метод или переменная доступны везде
`protected`	метод или переменная доступны в подклассах и в пределах пакета
`private`	метод или переменная доступны только в пределах класса
`(package)`	метод или переменная доступны только в пределах пакета
`synchronized`	метод, для которого осуществляется блокировка доступа к ресурсу
`transient`	переменная не должна быть сериализована

`native`	метод реализован на C или другим платформо-зависимым способом
`volatile`	переменная не должна оптимизироваться

Аннотации

Аннотации — это модификаторы, семантически не влияющие на программу, но содержащие метаданные, которые могут быть использованы при анализе кода, в процессе компиляции программы или во время её исполнения.

Стандартные аннотации языка Java:

`@Deprecated`	`/*`
	` * Этот метод устарел, заменён`
	` * на aBetterAlternative() и`
	` * не рекомендуется к использованию.`
	` */`
	`@Deprecated`
	`public void theDeprecatedMethod() {`
	`}`
	
	`public void aBetterAlternative() {`
	`}`

`@SupressWarnings`	`public static void main(String[] args) {`
	`  // Блокирует предупреждение компилятора`
	`  @SuppressWarnings("unchecked")`
	`  Collection<Integer> c =`
	`  new LinkedList();`
	`}`

`@Override`	`class BaseClass {`
	`  public void toBeOverriddenMethod() { }`
	`}`
	
	`public class ClassWithOverrideMethod {`
	`  // Переопределяет метод родительского`
	`  // класса`
	`  @Override`
	`  public void toBeOverriddenMethod() { }`
	`}`

Перечисляемые типы

Перечисляемый тип (enum) — это тип, значения которого ограничены набором констант.

```
public enum Season {
    WINTER,
    SPRING,
    SUMMER,
    AUTUMN
}

public class TestSeason {
    public static void main(String[] args) {
        Season s = Season.SUMMER;
        System.out.println("Current season is " + s);
    }
}
```

Использование объектов

объявление объекта (`a = null`)	`Point a;`
создание объекта (выделяется память)	`a = new Point(10,10);`
доступ к переменным	`a.x = 20;`
вызов методов	`a.show();`
уничтожение неиспользуемого объекта	"сборщик мусора"

метод `finalize()`:

```
protected void finalize() throws Throwable {
  super.finalize();
  if (file != null) {
    file.close();
    file = null;
  }
}
```

Пакеты, входящие в JDK 1.6

java.applet	Классы для реализации апплетов.
java.awt	Классы для реализации графического пользовательского интерфейса.
java.awt.color	Классы для раскраски компонентов пользовательского интерфейса.
java.awt.datatransfer	Классы для поддержки передачи информации внутри приложений и между ними.
java.awt.dnd	Классы для реализации механизма drag-n-drop в пользовательских интерфейсах.
java.awt.event	Классы и интерфейсы для обработки событий.
java.awt.font	Классы и интерфейсы, связанные со шрифтами.
java.awt.geom	Классы для генерации объектов двухмерной графики.
java.awt.im	Классы и интерфейсы для реализации ввода данных.
java.awt.image	Классы для обработки изображений.
java.awt.print	Классы и интерфейсы, реализующие механизм вывода данных на печать.
java.beans	API для модели компонентов JavaBeans.
java.io	Классы для различных потоков ввода-вывода, сериализации и работы с файловой системой.
java.lang	Базовые классы и интерфейсы языка Java.
java.lang.ref	Классы, обеспечивающие ряд возможностей по взаимодействию со «сборщиком мусора» виртуальной машины.
java.lang.reflect	Классы для проверки структуры классов и ее отражения.
java.math	Классы для чисел произвольной точности
java.net	Классы для поддержки сетевого взаимодействия.
java.nio	Классы, реализующие расширенные возможности ввода-вывода с использованием буферизованных контейнеров для различных типов данных.
java.rmi	Классы и интерфейсы для обеспечения удаленного вызова методов.
java.rmi.activation	API, реализующее возможности по активации RMI-объектов.
java.rmi.dgc	Классы и интерфейсы для реализации распределенной "сборки мусора".
java.rmi.registry	Классы для поддержки базы данных объектов и услуг.

java.rmi.server	Классы для обеспечения удаленного доступа со стороны сервера.
java.security	Классы и интерфейсы для обеспечения защиты данных от несанкционированного доступа.
java.sql	Стандартный интерфейс доступа к базам данных.
java.text	Классы и интерфейсы для обеспечения много-языковой поддержки
java.util	Вспомогательные классы, обеспечивающие работу со структурами данных и форматирование текста с учетом локализации.
java.util.concurrent	Классы, обеспечивающие расширенные возможности многопоточного программирования.
java.util.jar	Классы для работы с JAR-архивами.
java.util.logging	Классы и интерфейсы, реализующие журналирование исполнения программ.
java.util.prefs	API для работы с пользовательскими и системными конфигурационными параметрами.
java.util.regex	Классы для обработки данных с помощью регулярных выражений.
java.util.zip	Классы для обеспечения архивации.
javax.annotation.processing	Классы, реализующие разнообразные механизмы обработки аннотаций.
javax.crypto	Классы и интерфейсы для криптографических операций.
javax.imageio	API для ввода-вывода графических изображений.
javax.management	Классы и интерфейсы, реализующие API Java Management Extensions.
javax.naming	Классы и интерфейсы для доступа к службам имён.
javax.net	Дополнительные классы для поддержки сетевого взаимодействия.
javax.print	Классы и интерфейсы, реализующие вывод данных на печать.
javax.rmi	API для RMI-IIOP.
javax.script	Классы и интерфейсы, позволяющие использовать программы, написанные на скриптовых языках программирования.
javax.security.auth	Классы и интерфейсы, реализующие механизмы аутентификации и авторизации.
javax.security.cert	Классы и интерфейсы, реализующие поддержку сертификатов открытых ключей.
javax.security.sasl	Классы и интерфейсы, реализующие поддержку SASL.

javax.sound.midi	API для создания и воспроизведения MIDI-звуков.
javax.sound.sampled	Классы и интерфейсы для захвата, преобразования и воспроизведения дискретных звуковых данных.
javax.sql	API для доступа к базам данных.
javax.swing	Набор легковесных компонентов для создания графических интерфейсов пользователя.
javax.swing.border	Классы и интерфейсы для создания рамок вокруг компонентов Swing.
javax.swing.colorchooser	Классы и интерфейсы, используемые компонентом JColorChooser.
javax.swing.event	Классы событий, создаваемых компонентами Swing.
javax.swing.filechooser	Классы и интерфейсы, используемые компонентом JFileChooser.
javax.swing.plaf	Классы и интерфейсы, реализующие возможности изменения внешнего вида компонентов Swing.
javax.swing.table	Классы и интерфейсы для работы с компонентом javax.swing.JTable.
javax.swing.text	Классы и интерфейсы, реализующие редактируемые и нередактируемые текстовые компоненты.
javax.swing.tree	Классы и интерфейсы для работы с компонентом javax.swing.JTree
javax.swing.undo	API для реализации функций undo / redo («отменить» / «вернуть»).
javax.tools	Интерфейсы для вызова внешних утилит (например, компиляторов).
javax.transaction	Классы и интерфейсы, описывающие правила взаимодействия менеджеров транзакций и менеджеров ресурсов для разных протоколов.
javax.xml	Классы и интерфейсы, необходимые для работы с XML, а также с основанными на XML протоколами.
org.w3c.dom	Базовые интерфейсы для DOM (Document Object Model), использующиеся при обработке XML.
org.w3c.sax	Базовые классы и интерфейсы SAX API

Пакет java.lang

Основные классы и интерфейсы, входящие в пакет `java.lang` показаны на рис. 2.

Рисунок 2. Пакет java.lang

Класс Object

Класс **Object** лежит в основе всей иерархии классов Java

Методы класса:

`public final native Class getClass()`	возвращает класс объекта
`public final native void notify()`	пробуждает поток, ожидающий монитор объекта
`public final native void notifyAll()`	пробуждает все потоки, ожидающие монитор объекта
`public final native void wait()`	ждет оповещения другим потоком
`public native int hashCode()`	возвращает хэш-код объекта
`public boolean equals(Object obj)`	сравнивает объекты на равенство
`public native Object clone()`	возвращает копию объекта
`public String toString()`	преобразует объект в строку символов
`protected void finalize()`	вызывается сборщиком мусора при разрушении объекта

Класс Class

Экземпляры этого класса описывают классы, интерфейсы, массивы и примитивные типы данных работающего приложения. У этого класса нет конструкторов, объекты создаются либо динамически виртуальной машиной Java, либо с помощью метода `getClass()` любого объекта.

Методы:

`forName(String className)`	возвращает объект `Class` для заданного имени
`getName()`	возвращает имя класса
`newInstance()`	создает новый экземпляр класса
`getSuperclass()`	возвращает суперкласс
`isInterface()`	определяет, является ли объект интерфейсом
`getInterfaces()`	возвращает интерфейсы класса
`isArray()`	определяет, является ли объект массивом
`isPrimitive()`	определяет, является ли тип примитивным

Класс System

Содержит набор методов для доступа к системным функциям, а также переменные `in`, `out` и `err`, представляющие соответственно стандартные потоки ввода, вывода и ошибок.

`getProperty(String name)`	возвращает значение свойства с именем name
`getenv(String name)`	возвращает значение переменной окружения
`arraycopy(Object src, nt pos1, Object dst, int pos2, int n)`	копирует элементы массива в другой массив
`exit(int status)`	выполняет выход из программы
`gc()`	запускает сборщик мусора
`loadLibrary(String name)`	загружает динамическую библиотеку
`runFinalization()`	запускает методы `finalize()` объектов
`currentTimeMillis()`	возвращает миллисекунды с 1 января 1970 г.

Класс Math

Содержит константы и методы для реализации математических функций:

`E, PI`
`abs(x), max(a,b), min(a,b), round(x), rint(x), ceil(x), floor(x);`
`pow(x,y), exp(x), log(x), sqrt(x), IEEEremainder(x,y), random(x);`
`sin(x), cos(x), tan(x), asin(x), acos(x), atan(x), atan2(x,y).`

Классы-оболочки

Используются для объектного представления примитивных типов данных. Реализуют методы преобразования из примитивных типов и обратно, а также в строковое представление и обратно.

К классам-оболочкам относятся: `Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Void`.

Автоупаковка и автораспаковка

Допускается присваивать переменным примитивного типа значения соответствующих классов-оболочек и наоборот — в этом случае вызов метода преобразования будет автоматически добавлен компилятором:

`Integer answer =`
`    new Integer(42);`
 	
`int i = answer.intValue();`
	`Integer answer = 42;`
	
	`int i = answer;`

Класс Exception

Является предком всех классов исключений, сигнализирующих о нестандартных ситуациях, которые должны быть специальным образом обработаны. Исключения, которые может вызывать какой-либо метод должны объявляться в операторе `throws` этого метода (кроме исключений, порожденных от класса `RuntimeException`).

Классы исключений, входящие в состав пакета java.lang, приведены на рис. 3.

Рисунок 3. Исключения пакета java.lang

Класс RuntimeException

Данные исключения описывают исключения времени выполнения, которые могут возникать в любом месте программы, и которые не нужно объявлять в операторе `throws`.

Основные классы необрабатываемых исключений приведены на рис. 4.

Рисунок 4. Необрабатываемые исключения

Класс Error

Объекты `Error`, в отличие от исключений, не должны перехватываться, и обычно приводят к экстренному завершению программы.

Основные классы ошибок приведены на рис. 5.

Рисунок 5. Потомки класса Error

Пакет java.util

Основные классы и интерфейсы, входящие в состав пакета java.util показаны на рис. 6.

Рисунок 6: Пакет java.util

Классы и интерфейсы пакета java.util

`BitSet`	представляет битовый массив произвольной длины
`Date`	описывает значения даты и времени в миллисекундах
`Calendar`	работает с единицами измерения даты

`GregorianCalendar`	реализует стандартный Григорианский календарь
`TimeZone`	описывает временные пояса
`SimpleTimeZone`	временная зона для Григорианского календаря
`Random`	используется для генерации псевдослучайных чисел
`EventObject`	определяет базовый тип события
`EventListener`	является меткой слушателя событий
`Observer`	интерфейс для объектов-наблюдателей
`Observable`	базовый класс для всех наблюдаемых объектов
`Dictionary`	шаблон для создания ассоциативных массивов
`Properties`	используется для хранения списка системных свойств
`Locale`	описывает понятие местности
`ResourceBundle`	описывают набор локализованных ресурсов

Работа с текстом и многоязыковая поддержка

Класс String

Используется для представления символьных строк (констант).

Конструкторы:

```
public String()
public String(char chars[])
public String(char chars[],  int offset, int length)
public String(byte bytes[])
public String(byte bytes[],
              int offset, int length, String encoding)
public String(String str)
public String(StringBuffer buffer)
```

Методы:

`public int length()`	длина строки
`public char charAt(int index)`	символ в заданной позиции
`public boolean equals(Object o)`	сравнение строки с объектом
`public int compareTo(String s)`	сравнение со строкой
`public boolean startsWith(String s)`	истина, если строка начинается с префикса
`public boolean endsWith(String s)`	истина, если строка заканчивается суффиксом
`public int indexOf(int char)`	позиция символа
`public int indexOf(String str)`	позиция подстроки
`public int lastIndexOf(int char)`	позиция символа с конца
`public int lastIndexOf(String str)`	позиция подстроки с конца
`public static String valueOf(…)`	преобразование в строку

Класс StringBuffer

Используется для представления изменяемых строк.

Конструкторы:

```
public StringBuffer()
public StringBuffer(int length)
public StringBuffer(String str)
```

Методы:

`public int length()`	длина строки в буфере
`public char charAt(int index)`	символ в заданной позиции
`public int capacity()`	размер буфера
`public StringBuffer append(…)`	добавление в конец буфера
`public StringBuffer insert(…)`	вставка в буфер
`public StringBuffer reverse()`	инверсия строки
`public void setCharAt(int i, char c)`	установка символа в заданной позиции
`public String toString()`	преобразование в строку

Использование String и StringBuffer

```
class ReverseString {
  public static String reverse(String source) {
    int i, len = source.length();
    StringBuffer dest = new StringBuffer(len);
    for (i = (len - 1); i >= 0; i--) {
      dest.append(source.charAt(i));
    }
    return dest.toString();
  }
}
```

Все строковые константы в Java имеют тип `String`. Оператор + для объектов типа `String` выполняет операцию соединения (конкатенации). Если в выражении присутствует хотя бы один объект типа `String`, остальные объекты преобразуются в `String` с помощью метода `toString()`.

Класс StringTokenizer

Используется для разбиения строки на лексемы.

Конструкторы:

```
public StringTokenizer(String string)
public StringTokenizer(String string, String delimiters)
```

Методы:

```
public boolean hasMoreTokens();
public String nextToken();
public String nextToken(String newDelimiters);
```

Пример:

```
String sentence = "It's a sentence, it can be tokenized.";

StringTokenizer st =
    new StringTokenizer(sentence, " ,.!?;-\n\r");

while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

Интернационализация

Интернационализация - это процесс создания приложений таким образом, чтобы они легко адаптировались для различных языков и регионов без внесения конструктивных изменений.

Характеристики интернационализированного приложения:

- один и тот же код может работать в любой местности при условии добавления данных о локализации;
- приложение отображает текст на родном языке конечного пользователя;
- текстовые элементы не являются частью кода, а хранятся отдельно и запрашиваются динамически;
- поддержка новых языков не требует перекомпиляции;
- данные, зависящие от местности, такие как даты и денежные единицы, отображаются в соответствии с регионом и языком конечного пользователя;
- приложение может быть быстро и легко локализовано.

Локализация

Локализация - это процесс адаптации программного обеспечения для определенного региона или языка путем добавления специфических для данной местности компонентов и перевода текста.

Данные, зависящие от местности:

- текст;
- числа;
- денежные единицы;
- дата и время;

- изображения;
- цвета;
- звуки.

Пример программы

```
import java.util.*;

public class IntTest {

  static public void main(String args[]) {
    if (args.length != 2) {
      System.out.println
        ("Format: java IntTest lang country");
      System.exit(-1);
    }
    String language = new String(args[0]);
    String country = new String(args[1]);
    Locale loc = new Locale(language, country);
    ResourceBundle messages =
      ResourceBundle.getBundle("MessagesBundle", loc);
    System.out.println(messages.getString("greeting"));
    System.out.println(messages.getString("inquiry"));
    System.out.println(messages.getString("farewell"));
  }

}
```

Имя файла	***Содержимое***

`MessageBundle.properties`	`greeting = Hello!`
	`inquiry = How are you?`
	`farewell = Goodbye!`

`MessageBundle_ru_RU.properties`	`greeting = Привет!`
	`inquiry = Как дела?`
	`farewell = До свидания!`

Класс Locale

Представляет определенный географический, политический или культурный регион (местность).

Конструкторы:

```
public Locale (String language, // ISO 639
               String country)  // ISO 3166
public Locale (String language,
               String country,
               String variant)
```

Пример:

```
Locale current = new Locale("en", "US");
Locale loc = new Locale("ru", "RU", "koi8r");
```

Методы:

```
public String getLanguage()
public String getCountry()
public String getVariant()
public static Locale getDefault()
public static synchronized void setDefault(Locale loc)
```

Метод `Locale.getDefault()` возвращает значение `Locale`, используемое по умолчанию. Установить его можно следующим образом:

- с помощью системных свойств `user.language` и `user.region`
- с помощью метода `Locale.setDefault()`

Получить список возможных комбинаций языка и страны можно с помощью статического метода `getAvailableLocales()` различных классов, которые используют форматирование с учетом местных особенностей.

Например:

```
Locale list[] = DateFormat.getAvailableLocales()
```

Класс ResourceBundle

Абстрактный класс, предназначенный для хранения наборов зависящих от местности ресурсов. Обычно используется один из его подклассов:

- `ListResourceBundle`
- `PropertyResourceBundle`

Представляет собой набор связанных классов с единым базовым именем, и различающихся суффиксами, задающими язык, страну и вариант.

```
MessageBundle
MessageBundle_ru
MessageBundle_en_US
MessageBundle_fr_CA_UNIX
```

Методы:

```
public static final ResourceBundle getBundle(String name)
                                 throws MissingResourceException
public static final ResourceBundle getBundle(String name,
                                              Locale locale)
                                 throws MissingResourceException
```

Эти методы возвращают объект одного из подклассов `ResourceBundle` с базовым именем `name` и местностью, заданной объектом `locale` или взятой

по умолчанию. При отсутствии данного ресурса осуществляется поиск наиболее подходящего из имеющихся (путем последовательного исключения суффиксов), при неудаче инициируется исключение `MissingResourceException`.

Класс ListResourceBundle

Абстрактный класс, управляющий ресурсами с помощью списка. Используется путем создания набора классов, расширяющих ListResourceBundle, для каждой поддерживаемой местности и определения метода `getContents()`.

Пример:

```
public class MessageBundle_ru extends ListResourceBundle {

  public Object[][] getContents() {
    return contents;
  }

  public Object[][] contents = {
    { "greeting", "Привет!" },
    { "inquiry", "Как дела?" },
    { "farewell", "До свидания!" }
  };

}
```

Массив `contents` содержит список пар ключ-значение, причем ключ должен быть объектом типа `String`, а значение - `Object`.

Объект класса `ListResourceBundle` можно получить вызовом статического метода `ResourceBundle.getBundle()`.

```
ResourceBundle messages =
    ResourceBundle.getBundle("MessageBundle",
                             new Locale("ru", "RU"));
```

Поиск классов осуществляется в следующей последовательности:

1. `MessageBundle_ru_RU.class`
2. `MessageBundle_ru.class`
3. `MessageBundle.class`

Для получения желаемого значения объекта используется метод `getObject`:

```
String s = (String) messages.getObject("greeting");
```

Класс PropertyResourceBundle

Абстрактный класс, управляющий ресурсами с помощью набора свойств. Используется в случаях, когда локализуемые объекты имеют тип `String`.

Ресурсы хранятся отдельно от кода, поэтому для добавления новых ресурсов не требуется перекомпиляция.

Пример файла `Message_it.properties`:

```
greeting = Ciao!
inquiry = Come va?
farewell = Arrivederci!
```

Объект класса PropertyResourceBundle можно получить вызовом статического метода `ResourceBundle.getBundle()`:

```
ResourceBundle messages =
  ResourceBundle.getBundle("MessageBundle",
                           new Locale("it", "IT"));
```

Если метод `getBundle()` не может найти соответствующий класс, производится поиск файлов с расширением .properties в той же последовательности, как и для `ListResourceBundle`:

1. `MessageBundle_it_IT.properties`
2. `MessageBundle_it.properties`
3. `MessageBundle.properties`

Для получения значения свойства используется метод `getString`.

```
String s = messages.getString("greeting")
```

Иерархия классов java.text

Иерархия классов пакета java.text приведена на рис. 7.

Класс NumberFormat

Абстрактный класс, позволяющий форматировать числа, денежные единицы, проценты в соответствии с форматом, принятым в определенной местности*.

Форматирование осуществляется в 2 этапа:

---

* Список допустимых местностей, для которых определены форматы, можно получить с помощью статического метода
`public Locale[] NumberFormat.getAvailableLocales()`

Рисунок 7. Пакет java.text

1. Получение требуемого экземпляра класса с помощью одного из методов
    - `getNumberInstance`
    - `getCurrencyInstance`
    - `getPercentInstance`
2. Вызов метода `format()` для получения отформатированной строки.

Числа:

```
NumberFormat formatter =
    NumberFormat.getNumberInstance(Locale.GERMANY);
String result = formatter.format(123456.789);
```

Денежные единицы:

```
NumberFormat formatter =
    NumberFormat.getCurrencyInstance(Locale.FRANCE);
String result = formatter.format(4999.99);
```

Проценты:

```
NumberFormat formatter =
    NumberFormat.getPercentInstance(Locale.US);
String result = formatter.format(.75);
```

Класс DecimalFormat

Позволяет создавать собственные форматы для чисел, денежных единиц и процентов.

Порядок использования:

1. Вызывается конструктор с шаблоном в качестве аргумента

   ```
   String pattern = "###,##0.##";
   DecimalFormat formatter = new DecimalFormat(pattern);
   ```

2. Вызывается метод `format()` для получения отформатированной строки

   ```
   String s = formatter.format(123123.456);
   ```

Значения символов шаблона:

Символ	***Значение***
0	цифра
#	цифра, или пробел в случае нуля
.	десятичный разделитель
,	групповой разделитель
;	разделитель форматов
-	префикс отрицательного числа
%	процент (значение умножается на 100)
?	промилле (значение умножается на 1000)
¤	заменяется обозначением денежной единицы (международным если удвоен) и в формате вместо десятичного будет использован денежный разделитель
X	любой другой символ в префиксе или суффиксе
'	используется для экранирования специальных символов в префиксе или суффиксе

Класс DecimalFormatSymbols

Используется для изменения значения используемых по умолчанию разделителей в классе `DecimalFormat`.

Конструкторы:

```
public DecimalFormatSymbols()
public DecimalFormatSymbols(Locale locale)
```

Методы:

```
public void setZeroDigit(char c)
public void setGroupingSeparator(char c)
public void setDecimalSeparator(char c)
public void setPerMill(char c)
public void setPercent(char c)
public void setDigit(char c)
public void setNaN(char c)
public void setInfinity(char c)
public void setMinusSign(char c)
public void setPatternSeparator(char c)
```

Имеются соответствующие методы `get()` для получения установленных значений.

Для передачи значений разделителей объект `DecimalFormatSymbols` передается конструктору класса `DecimalFormat` в качестве аргумента:

```
DecimalFormatSymbols symbols = new DecimalFormatSymbols();
symbols.setGroupingSeparator(" ");
DecimalFormat formatter =
    new DecimalFormat("#,##0.00", symbols);
String s = formatter.format(4.50);
```

Класс DateFormat

Абстрактный класс, позволяющий форматировать дату и время в соответствии с форматом, принятым в определенной местности[*].

Форматирование осуществляется в 2 этапа:

1. Получение требуемого экземпляра класса с помощью одного из методов
    - `getDateInstance`
    - `getTimeInstance`
    - `getDateTimeInstance`

[*] Список допустимых местностей, для которых определены форматы, можно получить с помощью статического метода
`public Locale[] DateFormat.getAvailableLocales()`

2. Вызов метода `format()` для получения отформатированной строки.

Дата:

```
DateFormat formatter =
    DateFormat.getDateInstance(DateFormat.SHORT, Locale.UK);
String result = formatter.format(new Date());
```

Время:

```
DateFormat formatter =
    DateFormat.getTimeInstance(DateFormat.LONG,
                               Locale.FRANCE);
String result = formatter.format(new Date());
```

Дата и время:

```
DateFormat formatter =
    DateFormat.getDateTimeInstance(DateFormat.FULL,
                                   DateFormat.FULL,
                                   Locale.US);
String result = formatter.format(new Date());
```

Класс SimpleDateFormat

Позволяет создавать форматы для даты и времени.

Порядок использования:

1. Вызывается конструктор с шаблоном в качестве аргумента:
   ```
   SimpleDateFormat formatter =
       new SimpleDateFormat("K:mm EEE MMM d ''yy");
   ```
2. Вызывается метод `format()` для получения отформатированной строки:
   ```
   String s = formatter.format(new Date());
   ```

Шаблоны SimpleDateFormat

Символ	***Значение***	***Тип***	***Пример***
G	обозначение эры	текст	AD
y	год	число	1996
M	месяц года	текст/число	July или 07
d	число месяца	число	23
h	часы (1-12)	число	5
H	часы (0-23)	число	22
m	минуты	число	45
s	секунды	число	31
S	миллисекунды	число	978
E	день недели	текст	Tuesday
D	номер дня в году	число	189
F	день недели в месяце	число	2 (2^{nd} Wed in July)

w	неделя в году	число	27
W	неделя в месяце	число	2
a	знак AM/PM	текст	PM
k	часы (1-24)	число	24
K	часы (0-11)	число	0
z	временная зона	текст	GMT
'	символ экранирования		

Для текста 4 и более символов задают полную форму, 3 и меньше - сокращенную. Для чисел количество символов указывает минимальный длину числа (кроме года - 2 символа обозначают год, записанный 2-мя цифрами). Смешанный формат с 3 и более символами - текст, менее 3-х — число.

Класс DateFormatSymbols

Используется для изменения названий месяцев, дней недели и других значений в классе `SimpleDateFormat`.

Конструкторы:

```
public DateFormatSymbols()
public DateFormatSymbols(Locale locale)
```

Методы:

```
public void setEras(String newValue[])
public void setMonths(String newValue[])
public void setShortMonths(String newValue[])
public void setWeekDays(String newValue[])
public void setShortWeekDays(String newValue[])
public void setAmPmStrings(String newValue[])
public void setZoneStrings(String newValue[])
public void setPatternChars(String newValue[])
```

Имеются соответствующие методы `get()` для получения установленных значений.

Для передачи значений разделителей объект `DateFormatSymbols` передается конструктору класса `SimpleDateFormat` в качестве аргумента:

```
DateFormatSymbols symbols = new DateFormatSymbols();
String weekdays[] = {"Пн","Вт","Ср","Чт","Пт","Сб","Вс"};
symbols.setShortWeekDays(weekdays);
DecimalFormat formatter = new SimpleDateFormat("E",
symbols);
String s = formatter.format(new Date());
```

Класс MessageFormat

Используется для выдачи сообщений на различных языках с включением изменяющихся объектов.

Использование класса:

1. Выделение переменных объектов в сообщении:

   At **1:15 PM** on **April 13, 1998**, we detected 7 spaceships on the planet **Mars**.

2. Помещение шаблона сообщения в ResourceBundle:

```
ResourceBundle messages =
    ResourceBundle.getBundle("MessageBundle",
                             currentLocale);
```

   содержимое файла `MessageBundle.properties`:

```
template = At {2,time,short} on {2,date,long}, we
detected {1,number,integer} spaceships on the planet {0}.
planet = Mars
```

3. Установка аргументов сообщения:

```
Object[] args = {
    messages.getString("planet"),
    new Integer(7),
    new Date()
}
```

4. Создание объекта `MessageFormat`:

```
MessageFormat formatter =
    new MessageFormat(messages.getString("template"));
formatter.setLocale(currentLocale);
```

5. Форматирование сообщения:

```
String s = formatter.format(args);
```

Синтаксис аргументов MessageFormat

```
{ индекс аргумента, [ тип, [ стиль ] ] }
```

Индекс задает порядковый индекс аргумента в массиве объектов (0-9).

Типы и стили аргументов:

Возможные типы	***Возможные стили***
number	currency, percent, integer, шаблон числа
date	short, long, full, medium, шаблон даты
time	short, long, full, medium, шаблон времени
choice	шаблон выбора

Если тип и стиль отсутствуют, то аргумент должен являться строкой. При отсутствии стиля, он принимается по умолчанию.

Класс ChoiceFormat

Используется для задания возможности выбора различных элементов в зависимости от значения параметров.

Использование класса:

1. Выделение переменных объектов в сообщении:

```
There are no files on disk C.
There is one file on disk C.
There are 3 files on disk C.
```

2. Помещение шаблона сообщения в ResourceBundle:

   содержимое файла `MessageBundle.properties`:

```
template = There {0} on disk {1}.
no = are no files
one = is one file
many = are {2} files
```

3. Создание объекта `ChoiceFormat`:

```
double limits[] = {0,1,2}
String choices[] = { messages.getString("no"),
                     messages.getString("one"),
                     messages.getString("many") }
ChoiceFormat choice = new ChoiceFormat(limits, choices);
```

4. Создание объекта `MessageFormat`:

```
MessageFormat formatter =
    new MessageFormat(messages.getString("template"));
formatter.setLocale(currentLocale);
Format[] formats = { choiceForm, null,
                     NumberFormat.getInstance() }
formatter.setFormats(formats);
```

5. Установка аргументов сообщения:

```
Object[] args = { 1, "C", 1 };
```

6. Форматирование сообщения

```
String s = formatter.format(args);
```

Класс Collator

Используется для выполнения сравнений строк в различных языках.

Получение объекта:

```
Collator c = Collator.getInstance(Locale.US);
```

Методы:

```
public int compare(String s1, String s2)
public void setStrength(int value) // value = PRIMARY,
                                   //         SECONDARY
                                   //         TERTIARY
                                   //         IDENTICAL
public void setDecomposition(int value)
                  // value = NO_DECOMPOSITION
                  //         CANONICAL_DECOMPOSITION
                  //         FULL_DECOMPOSITION
```

Пример:

```
c.setStrength(PRIMARY);
if (c.compare("ABC", "abc") < 0) {
    // "ABC" < "abc"
} else {
    // "ABC" >= "abc"

}
```

Класс RuleBasedCollator

Используется для задания собственных правил сортировки символов.

Набор правил передается конструктору в виде строки символов:

```
String rule = "a < b < c < d";
RuleBasedCollator c = new RuleBasedCollator(rule);
```

Формат правил:

<	следующий символ больше предыдущего
;	следующий символ больше предыдущего без учета акцентов
,	следующий символ больше предыдущего без учета размера букв
=	следующий символ равен предыдущему
@	сортировка акцентов в обратном порядке
&	дополнительное правило для уже встречавшихся символов

Пример:

```
RuleBasedCollator us =
    (RuleBasedCollator) Collator.getInstance(Locale.US);
String rule = us.getRules();
String extraRule = "& c,C < ch,CH";
RuleBasedCollator sp =
    new RuleBasedCollator(rule + extraRule);
```

```
String words[] = {"canoe", "corozon", "chiquita"};
String tmp;

for (int i = 0; i < words.length; i++) {
  for (int j = i+1; j < words.length; j++) {
    if (sp.compare(words[i], words[j]) > 0 {
      tmp = words[i];
      words[i] = words[j];
      words[j] = tmp;
    }
  }
}

System.out.println("" + words);
```

Класс CollationKey

Используется для повышения производительности при многочисленных сравнениях строк.

Пример:

```
Collator us = Collator.getInstance(Locale.US);
String words[] = {"apple", "orange", "lemon", "peach"};
String tmp;
CollationKey[] keys = new CollationKey(words.length);

for (int i = 0; i < words.length; i++) {
  keys[i] = us.getCollationKey(words[i]);
}

for (int i = 0; i < keys.length; i++) {
  for (int j = i+1; j < keys.length; j++) {
    if (keys[i].compareTo(keys[j])) > 0 {
      tmp = keys[i];
      keys[i] = keys[j];
      keys[j] = tmp;
    }
  }
}

for (CollationKey key : keys) {
  System.out.println(key.getSourceString() + "  ");
}
```

Использование легковесных процессов

Данный раздел посвящен работе с потоками (в некоторых других источниках они получили название легковесные процессы). Поток – отдельная выполняемая последователь-ность в программе. Фактически поток – это способ реализации многозадачности в Java.

Потоки используются при решении многих задач:

- анимация;
- воспроизведение и обработка звуковых данных;
- обновление и восстановление информации в фоновом режиме;
- ожидание и обработка информации, поступающей по сети.

При запуске поток получает в свое распоряжение определенную долю ресурсов процессора и, в дальнейшем, работает уже с ней. Говорят, что поток имеет "тело", которое содержит операторы, оно находится в методе `run()`.

Реализация потока

Java предусматривает две возможности реализации тела потока:

- для класса определяется интерфейс `Runnable` и используется метод `run()`, а также методы `start()`, `stop()`;
- постоение класса как потомка класса `java.lang.Thread`.

Создание потока

Метод `start()` выполняется каждый раз при обращении к странице, содержащей апплет. Метод проверяет состояние объекта `Thread`, которая должна быть описана выше, если значение переменной `null`, то создается новый объект типа `Thread` и вызывается метод `start()` класса `Thread`.

```
Thread myNewThread;

public void start() {
  if (myNewThread == null) {
    myNewThread = new Thread(this, "MyNewThreadName");
    myNewThread.start();
  }
}
```

Рассмотрим процесс создания нового объекта `Thread`. Наибольший интерес для нас представляет первый аргумент в вызове конструктора `Thread`. Переменная `this` указывает на текущий апплет. Этот аргумент должен

реализовать интерфейс Runnable, после чего он становится адресатом потока. Второй аргумент определяет имя потока.

Остановка потока

Когда вы покидаете страницу с апплетом, использующим потоки, вызывается метод stop(), основное назначение которого является присвоение объекту типа Thread значения null

```
public void stop() {
  myNewThread = null;
}
```

Возможен и другой вариант, а именно непосредственный вызов метода stop

```
myNewThread.stop();
```

класса Thread, но он не всегда приемлем, что связано со следующим: метод stop может быть вызван в "неудачный" момент выполнения метода run(), в результате чего поток может не прекратить свое выполнение. При повторном посещении страницы будет вызван метод start и апплет начнет свое выполнение.

Выполнение потока

Метод run() является "сердцем" потока, именно он определяет назначение не только потока, но и, зачастую, всего класса.

```
public void run() {
  while (Thread.currentThread() == clockThread){
    repaint();
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {}
  }
}
```

Как было показано выше, перед остановкой работы апплета, значение потока становится равным null. Метод run() будет выполняться, пока поток не примет значение null. В данном примере апплет перерисовывает сам себя, затем выполнение потока приостанавливается на 1 секунду (1000 милисекунд). При вызове repaint() происходит обращение к методу paint(), который меняет содержимое экрана.

```
public void paint(Graphics g) {
  Date now = new Date();
  g.drawString(now.getHours() + ":" + now.getMinutes() +
               ":" + now.getSeconds(), 5, 10);
}
```

Состояние потока

Новый поток

Данное выражение создает новый пустой поток:

```
Thread myNewThread = new Thread();
```

С таким потоком вы можете совершить следующие действия: запустить или остановить (`stop()`, `start()`). Попытка вызвать любой другой метод работы с потоком приводит к вызову исключения `IllegalThreadStateException`.

Выполняемый

```
Thread myNewThread = new Thread();
myNewThread.start();
```

Метод `start()` подготавливает все необходимые ресурсы для запуска потока и передает управление методу `run()`.

Невыполняемый

Поток является невыполняемым, если находится в одном из четырех состояний:

- вызван метод `sleep`
- вызван метод `suspend`
- вызван метод `wait`
- поток заблокирован вводом/выводом

```
try {
  Thread.sleep(10000);
} catch (InterruptedException e) {
}
```

Поток прекратил свое выполнение на 10 секунд. После непродолжительного сна поток возобновляет свою работу. Вызов метода `resume()` для спящего потока не дает результатов.

Чтобы "разбудить" спящий поток (`sleep()`) необходим процесс ожидания, для возобновления работы потока (`suspend()`) достаточно вызвать метод `resume()`. Если поток ожидает установки какой-то переменной, то прервать это можно при помощи `notify` или `notifyAll`. Если блокировка по причине ожидание окончания ввода/вывода, то необходимо дождаться конца обмена.

Смерть потока

Смерть наступает в двух случаях:

- после выполнения метода `stop()`;
- по естественным причинам, т.е., когда метод `run()` завершил выполнение.

Метод `isAlive()` возвращает `true`, если поток отработал метод `start()` и не выполнил метод `stop()`. Если же возвращен `false`, то это либо новый поток, либо он умер.

Распределение приоритета между потоками

В классе `java.lang.Thread` описаны три идентификатора, определяющие приоритеты для потоков.

- `MIN_PRIORITY`
- `NORM_PRIORITY`
- `MAX_PRIORITY`

При создании потока ему по умолчанию устанавливается `NORM_PRIORITY`, изменить приоритет можно путем использования метода `setPriority(int)`. Изменение приоритета потока заключается в изменении выделяемого ему промежутка времени.

Использование нескольких потоков

Рассмотрим пример использования нескольких потоков в рамках одного приложения.

```
import java.applet.Applet;

class CreatTwoThreads extends Applet {
  public void CreatTwoThreads () {
    new CreatThread1().start();
    new CreatThread2().start();
  }
}

class CreatThread1 extends Thread {
  ...
  public void run () {
    ...
  }
}
class CreatThread2 extends Thread {
  ...
  public void run () {
    ...
  }
}
```

Таким образом, головной класс создает два новых объекта и запускает их на выполнение. Два вновь созданных класса являются независимыми

полноправными потоками и все действия, описанные в методе `run()` будут выполняться как потоковые.

Класс java.lang.ThreadGroup

Класс предназначен для объединения потоков в группы, что, в значительной степени, упрощает работу с потоками и позволяет более гибко управлять их работой. С группой потоков возможны те же основные операции, что и с простым потоком:

- запуск;
- останов;
- установка приоритетов;
- и т.д.

К тому же для группы потоков можно определять как родителя, так и потомков.

Методы класса java.lang.Thread

Условно разделим все методы на те, которые возвращают значения, и те, которые их установливают.

Первая группа:

- `activeCount()` возвращает текущее число активных потоков в группе;
- `currentThread()` возвращает ссылку на текущий выполняющийся поток;
- `getName()` возвращает имя потока;
- `getPriority()` возвращает приоритет потока;
- `getThreadGroup()` возвращает ссылку на группу, к которой принадлежит поток;
- `interrupted()` возвращает, является ли поток остановленным;
- `isAlive()` возвращает, жив ли поток ;
- `isDaemon()` возвращает, является поток демоном;
- `isInterrupted()` возвращает, остановлен ли поток.

Вторая группа:

- `setDaemon(boolean)` делает поток демоном;
- `setName(String)` устанавливает имя потока;
- `setPriority(int)` изменение приоритета потока.

Основы сетевого взаимодействия

Пакет java.net содержит классы, которые отвечают за различные аспекты сетевого взаимодействия. Классы можно поделить на 2 категории:

- низкоуровневый доступ (адреса, сокеты, интерфейсы)
- высокоуровневый доступ (URI, URL, соединения).

Классы, входящие в java.net, позволяют организовать передачу данных с помощью протоколов TCP, UDP, HTTP.

TCP и UDP являются протоколами транспортного уровня. Протоколы имеют следующие особенности.

Протокол	***TCP***	***UDP***
Установление соединения	да	нет
Подтверждение доставки сообщений	да	нет

При этом протокол TCP имеет более высокую надежность доставки сообщений, а UDP – более высокую скорость передачи данных.

Работа с адресами

Адреса хостов в локальной и глобальной сети представляются в виде последовательности чисел, которые получили название *IP-адреса*. IP-адрес может быть представлен в двух форматах — IPv4 и IPv6.
Адрес формата IPv4 имеет длину 32 бита, и представляется в виде четырех десятичных чисел от 0 до 255, разделенных точками (192.168.0.1 или 92.123.155.81).
Адрес формата IPv6 имеет длину 128 бит, и представляется в виде восьми 16-ричных чисел от 0 до FFFF, разделенных двоеточиями (1080:0:0:0:8:800:200C:417A).
Пользователям удобнее иметь дело с именами хостов, представленных в алфавитно-цифровом виде. Для преобразования цифровых адресов в алфавитно-цифровые имена используется служба имен *DNS (Domain Name Service)*.

Для представления IP-адресов и доступа к DNS в Java используется класс `InetAddress`. Данный класс имеет два подкласса – `Inet4Address` и

Inet6Address, но они используются редко, так как для большинства приложений хватает функциональности базового класса. Объект класса InetAddress содержит IP-адрес и имя хоста.

Экземпляр класса InetAddress можно получить с помощью статических методов класса:

`getLocalHost()`	Возвращает локальный хост
`getByAddress(String host, byte[] addr)`	Возвращает InetAddress с заданным IP-адресом и именем (корректность имени для данного адреса не проверяется)
`getByAddress(byte[] addr)`	Возвращает InetAddress с заданным IP-адресом
`getByName(String host)`	Возвращает InetAddress с заданным именем хоста (путем обращения к DNS)
`getAllByName(String host)`	Возвращает массив IP-адресов хоста с заданным именем (путем обращения к DNS)

Основные методы класса InetAddress:

`byte[] getAddress()`	Возвращает IP-адрес хоста
`String getHostName()`	Возвращает имя хоста

Передача данных по протоколу TCP

Для обеспечения передачи данных по протоколу TCP основным классом является java.net.Socket.

Конструктор класса Socket:

`Socket(InetAddress address, int port)`	создает соединение и подключает его к заданному порту по заданному IP-адресу

Методы класса Socket:

`InputStream getInputStream()`	возвращает входной поток данных
`OutputStream getOutputStream()`	возвращает выходной поток данных
`void setSoTimeout(int ms)`	устанавливает время ожидания

	завершения операции чтения из входного потока сокета в милли-секундах. По истечении данного времени выбрасывается исключение `SocketTimeoutException`.
`void close()`	закрывает сокет

Для реализации сервера, который ожидает запрос от клиента и отвечает на него, используется класс `ServerSocket`.

Конструктор класса `ServerSocket`:

`ServerSocket(int port)`	создает подключение на заданном порту

Методы класса `ServerSocket`:

`Socket accept()`	ожидает соединение и устанавливает его.
`void setSoTimeout(int ms)`	устанавливает время ожидания установления соединения в миллисекундах. По истечении данного времени выбрасывается исключение `SocketTimeoutException`
`void close()`	закрывает сокет

Последовательность создания TCP-соединения на стороне клиента:

1	Получение объекта `InetAddress`	`InetAddress ia =` `  InetAddress.getLocalHost();`
2	Создание сокета. При этом задается адрес (объект `InetAddress`) и порт, к которым будет устанавли-ваться соединение	`Socket soc =` `  new Socket(ia, 8888);`

3	Получение входного и выход-ного потоков сокета	`InputStream is =`
		`  soc.getInputStream();`
		`OutputStream os =`
		`  soc.getOutputStream();`

4	Чтение данных из входного и запись данных в выходной поток	`is.read()` `os.write()`

5	Закрытие потоков	`is.close();` `os.close();`
6	Закрытие сокета	`soc.close();`

Последовательность создания TCP-соединения на стороне сервера:

1	Создание объекта `ServerSocket`, который будет принимать соединения на заданный порт	`ServerSocket ss = new ServerSocket(8888);`
2	Ожидание соединения от клиента и получение сокета для коммуникации с клиентом.	`Socket soc = ss.accept();`
3	Получение входного и выходного потоков сокета	`InputStream is = soc.getInputStream();` `OutputStream os = soc.getOutputStream();`
4	Чтение данных из входного и запись данных в выходной поток	`is.read()` `os.write()`
5	Закрытие потоков	`is.close();` `os.close();`
6	Закрытие сокета	`soc.close();`

Передача данных по протоколу UDP

При работе с UDP используются следующие основные классы: `java.net.DatagramPacket` (представляющий передаваемое сообщение — датаграмму) и `java.net.DatagramSocket`.

Конструкторы класса `DatagramPacket`:

`DatagramPacket(byte[] buf, int length)`	создает датаграмму из задан-ного массива байтов с задан-ной длиной
`DatagramPacket(byte[] buf, int length, InetAddress addr, int port)`	создает датаграмму из задан-ного массива байтов с задан-ной длиной, для отправления на заданный адрес по задан-ному порту

Методы класса `DatagramPacket`:

`InetAddress getAddress()`	возвращает адрес
`int getPort()`	возвращает порт

`byte[] getData()`	возвращает массив данных
`int getLength()`	возвращает длину данных

Конструкторы класса `DatagramSocket`:

`DatagramSocket()`	создает сокет с использова-нием первого доступного локального порта
`DatagramSocket(int port)`	создает сокет с использова-нием заданного порта

Методы класса `DatagramSocket`:

`void send(DatagramPacket p)`	отсылает заданную датаграмму
`void receive(DatagramPacket p)`	принимает данные в заданную датаграмму
`void setSoTimeout(int ms)`	устанавливает время ожидания завершения операции приема датаграммы в миллисекундах. По истечении данного времени создается исключение `SocketTimeoutException`
`void close()`	закрывает сокет

Последовательность создания UDP-сокета на стороне сервера:

1	Создание объекта `DatagramSocket`, который будет принимать соединения на заданный порт и буферов для ввода и вывода	`DatagramSocket ds = new DatagramSocket(7777);`
		`byte[] ib = new byte[256];`
		`byte[] ob = new byte[256];`

2	Получение датаграммы от клиента	`DatagramPacket ip = new DatagramPacket(ib, ib.length);` `ds.receive(ip);`
3	Формирование ответа клиенту в виде массива байтов `ob`	

4	Формирование датаграммы и отсылка ее клиенту (адрес и порт клиента получаются из полученной от клиента датаграммы)	`InetAddress addr =`
		`  ip.getAddress();`
		`int port = ip.getPort();`
		`DatagramPacket op =`
		`  new DatagramPacket(ob,`
		`  ob.length, addr, port);`
		`ds.send(dp);`

5	Закрытие сокета	`ds.close();`

Последовательность создания UDP-сокета на стороне клиента:

1	Создание объекта `DatagramSocket` (без указания порта) и буферов для ввода и вывода	`DatagramSocket ds =`
		`  new DatagramSocket();`
		`byte[] ib = new byte[256];`
		`byte[] ob = new byte[256];`

2	Формирование запроса серверу в виде массива байтов `ob`	

3	Формирование датаграммы и отсылка ее серверу	`DatagramPacket op =`
		`  new DatagramPacket(ob,`
		`  ob.length, InetAddress`
		`  .getByName(`*`server_name`*`),`
		`  7777);`
		`ds.send(dp);`

4	Получение ответной датаграммы от сервера	`DatagramPacket ip =`
		`  new DatagramPacket(ib,`
		`  ib.length);`
		`ds.receive(ip);`

5	Закрытие сокета	`ds.close();`

Работа с URL-соединениями

URI (Unifrom Resource Identifier) – унифицированный идентификатор ресурса. Представляет собой символьную строку, идентифицирующую какой-либо ресурс (обычно ресурс Интернет).
URL (Uniform Resource Locator) – унифицированный локатор ресурса. Представляет собой подкласс URI, который кроме идентификации дает информацию о местонахождении ресурса.

Формат URL в общем виде (элементы в квадратных скобках могут отсутствовать):

```
[протокол:][//[логин[:пароль]@]хост[:порт]][путь]
[?запрос][#фрагмент]
```

Пакет `java.net` содержит классы URI и URL для представления URI и URL соответственно, класс `URLConnection`, использующийся для создания соединения с заданным ресурсом, и его подкласс `HttpURLConnection`, реализующий соединение с ресурсом по протоколу HTTP.

Рекомендуемая последовательность работы с URL-соединениями:

1. Создать URI:

```
URI uri =
  new URI("http://joe:12345@mail.ru:8080/index.php"
          + "?getMail=1&page=2#end");
URI uri2 = new URI("http", ”joe:12345”, ”mail.ru”,
  8080, ”index.php”,”getMail=1&page=2”,”end”);
```

2. Преобразовать URI в URL:

```
URL url = uri.toURL();
```

3. а) Открыть URL-соединение и поток данных:

```
URLConnection uc = url.openConnection();
uc.connect();
InputStream is = uc.getInputStream();
```

   б) Открыть поток данных:

```
InputStream is = url.openStream();
```

4. Получить данные;
5. Закрыть поток и соединение:

```
is.close(); uc.close();
```

Использование `URLConnection` по сравнению с простым открытием потока из URL позволяет дополнительно устанавливать параметры соединения, такие как возможность взаимодействия с пользователем, разрешение записи и чтения, а также получать информацию о соединении, такую, как даты создания и модификации, тип, длину и кодировку содержимого.

Работа с потоками ввода-вывода

Поток данных (stream) представляет из себя абстрактный объект, предназначенный для получения или передачи данных единым способом, независимо от связанного с потоком источника или приемника данных.

Иерархия потоков в Java

Потоки реализуются с помощью классов, входящих в пакет `java.io`. Потоки делятся на две больших группы — потоки ввода, и потоки вывода. Потоки ввода связаны с источниками данных, потоки вывода — с приемниками данных. Кроме того, потоки делятся на байтовые и символьные. Единицей обмена для байтовых потоков является байт, для символьных — символ Unicode.

	Потоки ввода	***Потоки вывода***
байтовые	`InputStream`	`OutputStream`
символьные	`Reader`	`Writer`

Кроме этих основных потоков, в пакет входят специализированные потоки, предназначенные для работы с различными источниками или приемниками данных, а также преобразующие потоки, предназначенные для преобразования информации, поступающей на вход потока, и выдачи ее на выход в преобразованном виде.

Класс InputStream

Представляет абстрактный входной поток байтов и является предком для всех входных байтовых потоков.

Конструктор:

`InputStream()`	Создает входной байтовый поток

Методы:

`abstract int read() throws IOException`	Читает очередной байт данных из входного потока. Значение должно быть от 0 до 255. При достижении конца потока возвращается -1. При ошибке ввода-вывода генерируется исключение. Подклассы должны обеспечить реализацию

	данного метода.
`int read(byte[] buf)`	Читает данные в буфер и возвращает количество прочитанных байтов.
`Int read(byte[] buf, int offset, int len)`	Читает не более `len` байтов в буфер, заполняя его со смещением `offset`, и возвращает количество прочитанных байтов
`void close()`	Закрывает поток.
`int available()`	Возвращает количество доступных на данный момент байтов для чтения из потока.
`long skip(long n)`	Пропускает указанное количество байтов из потока.
`boolean markSupported()`	Проверка на возможность повторного чтения из потока.
`void mark(int limit)`	Устанавливает метку для последующего повторного чтения. `limit` – размер буфера для операции повторного чтения.
`void reset()`	Возвращает указатель потока на предварительно установленную метку. Дальнейшие вызовы метода `read()` будут снова возвращать данные, начиная с заданной метки.

Класс OutputSrteam

Представляет абстрактный выходной поток байтов и является предком для всех выходных байтовых потоков.

Конструктор:

`OutputStream()`	Создает выходной байтовый поток.

Методы:

`abstract void write(int n) throws IOException`	Записывает очередной байт данных в выходной поток. Значащими являются 8 младших битов, старшие - игнорируются. При ошибке ввода-вывода генерируется исключение. Подклассы должны обеспечить реализацию данного метода.

`void write(byte[] buf)`	Записывает в поток данные из буфера.
`void write(byte[] buf, int offset, int len)`	Записывает в поток `len` байтов из буфера, начиная со смещения `offset`.
`void close()`	Закрывает поток.
`void flush()`	Заставляет освободить возможный буфер потока, отправляя на запись все записанные в него данные.

Класс Reader

Представляет абстрактный входной поток символов и является предком для всех входных символьных потоков.

Конструктор:

`Reader()`	создает входной символьный поток

Методы:

`abstract int read() throws IOException`	Читает очередной символ Unicode из входного потока. При достижении конца потока возвращается -1. При ошибке ввода-вывода генерируется исключение. Подклассы должны обеспечить реализацию данного метода.
`int read(char[] buf)`	Читает данные в буфер и возвра-щает количество прочитанных сим-волов.
`Int read(char[] buf, int offset, int len)`	Читает не более `len` символов в буфер, заполняя его со смещением `offset`, и возвращает количество прочитанных символов
`void close()`	закрывает поток
`int available()`	возвращает количество доступных на данный момент символов для чтения из потока
`long skip(long n)`	пропускает указанное количество символов из потока

`boolean markSupported()`	проверка на возможность повторного чтения из потока
`void mark(int limit)`	устанавливает метку для последую-щего повторного чтения. `limit` – размер буфера для операции повторного чтения
`void reset()`	возвращает указатель потока на предварительно установленную метку. Дальнейшие вызовы метода `read()` будут снова возвращать данные, начиная с заданной метки.

Класс Writer

Представляет абстрактный выходной поток символов и является предком для всех выходных символьных потоков.

Конструктор:

`Writer()`	создает выходной символьный поток

Методы:

`abstract void write(int n) throws IOException`	Записывает очередной символ Unicode в выходной поток. Знача-щими являются 16 младших битов, старшие - игнорируются. При ошибке ввода-вывода генерируется исклю-чение. Подклассы должны обеспе-чить реализацию данного метода.
`void write(char[] buf)`	Записывает в поток данные из буфера.
`void write(char[] buf, int offset, int len)`	Записывает в поток `len` символов из буфера, начиная со смещения `offset`
`void close()`	закрывает поток
`void flush()`	заставляет освободить возможный буфер потока, отправляя на запись все записанные в него данные.

Специализированные потоки

В пакет `java.io` входят потоки для работы со следующими основными типами источников и приемников данных:

тип данных	*байтовые*		*символьные*	
	входной	*выходной*	*входной*	*выходной*
файл	`FileInput-Stream`	`FileOutput-Stream`	`FileReader`	`FireWriter`
массив	`ByteArray-InputStream`	`ByteArray-OutputStream`	`CharArray-Reader`	`CharArray-Writer`
строка	-	-	`StringReader`	`String-Writer`
конвейер	`PipedInput-Stream`	`PipedOutput-Stream`	`PipedReader`	`Piped-Writer`

Конструкторы этих потоков в качестве аргумента принимают ссылку на источник или приемник данных — файл, массив, строку. Методы для чтения и записи данных — `read()` для входных потоков, `write()` для выходных потоков. Конвейер имеет особенность, что источником данных для входного конвейера является выходной конвейер, и наоборот. Обычно конвейеры используются для обмена данными между двумя потоками выполнения (`Thread`).

Пример чтения данных из файла:

```
FileReader f = new FileReader("myfile.txt");
char[] buffer = new char[512];
f.read(buffer);
f.close();
```

Преобразующие потоки

Этот тип потоков выполняет некие преобразования над данными других потоков. Конструкторы таких классов в качестве аргумента принимают поток данных.

Классы `BufferedInputStream`, `BufferedOutputStream`, `BufferedReader` и `BufferedWriter` предназначены для буферизации ввода-вывода. Они позволяют читать и записывать данные большими блоками. При этом обмен данными со стороны приложения ведется с буфером, а по мере необходимости в буфер из источника данных подгружается новая порция данных, либо из буфера данные переписываются в приемник данных. Класс `BufferedReader` имеет дополнительный метод `readLine()` для чтения строки символов, ограниченной разделителем строк. Класс `BufferedWriter` имеет дополнительный метод `newLine()` для вывода разделителя строк.

Классы `InputStreamReader` и `OutputStreamWriter` предназначены для преобразования байтовых потоков в символьные и наоборот. Кодировка задается в конструкторе класса. Если она опущена, то используется системная

кодировка, установленная по умолчанию). В конструктор класса `InputStreamReader` передается как аргумент объект класса `InputStream`, а в конструктор класса `OutputStreamWriter` – объект класса `OutputStream`. Методы `read()` и `write()` этих классов аналогичны методам классов `Reader` и `Writer`.

Пример использования:

Вариант 1:

```
FileInputStream f = new FileInputStream("myfile.txt");
InputStreamReader isr = new InputStreamReader(f);
BufferedReader br = new BufferedReader(isr);
br.readLine();
```

Вариант 2:

```
BufferedReader br = new BufferedReader(
                    new InputStreamReader(
                    new FileInputStream("myfile.txt")));
br.readLine();
```

`f` – поток байтов из файла myfile.txt;

`isr` – поток символов, преобразованный из байтового с учетом системной кодировки;

`br` – поток символов с поддержкой буферизации.

Классы `DataInputStream` и `DataOutputStream` предназначены для записи и чтения примитивных типов данных и содержат методы `readBoolean()`, `readInt()`, `readDouble()`, `writeFloat()`, `writeByte()` и другие подобные методы. Для успешного чтения таких данных из потока `DataInputStream` они должны быть предварительно записаны с помощью соответствующих методов `DataOutputStream` в том же порядке.

Классы `PrintStream` и `PrintWriter` предназначены для форматированного вывода в поток вывода. В них определено множество методов `print()` и `println()` с различными аргументами, которые позволяют напечатать в поток аргумент, представленный в текстовой форме (с использзованием системной кодировки). В качестве аргумента может использоваться любой примитивный тип данных, строка и любой объект. Методы `println` добавляют в конце разделитель строк.

Стандартные потоки ввода-вывода

Класс `java.lang.System` содержит 3 поля, представляющих собой стандарные консольные потоки:

поле	***класс***	***поток***	***по умолчанию***
`System.in`	`InputStream`	стандартный поток ввода	клавиатура
`System.out`	`PrintStream`	стандартный поток вывода	окно терминала
`System.err`	`PrintStream`	стандартный поток ошибок	окно терминала

Имеется возможность перенаправлять данные потоки с помощью методов `System.setIn`, `System.setOut`, `System.setErr`.

Пример чтения данных с клавиатуры и вывода в окно терминала:

```
BufferedReader br = new BufferedReader(
                    new InputStreamReader(System.in));
String s = br.readLine();
System.out.println("Введена строка : " + s);
System.out.println("Длина строки : " + s.length);
```

Сериализация объектов

Сериализация объектов - запись объекта со всеми полями и ссылками на другие объекты в виде последовательности байтов в поток вывода с последующим воссозданием (десериализацией) копии этого объекта путем чтения последовательности байтов сохраненного объекта из потока ввода.

Интерфейс java.io.Serializable

Интерфейс-метка, указывающий на то, что реализующий его класс может быть сериализован. Поля класса, не требующие сериализации, должны иметь модификатор `transient`.

Класс java.io.ObjectOutputStream

Предназначен для записи в поток вывода примитивных типов, подобно классу `DataOutputStream` и объектов (иерархически).

Конструктор класса `ObjectOutputStream`:

`ObjectOutputStream(OutputStream o)`	Создает объект класса, связанный с выходным потоком `o`.

Методы класса `ObjectOutputStream`:

`void writeObject(Object obj)`	Иерархически записывает в поток заданный объект.
`void useProtocolVersion(int v)`	Задает версию протокола сериализа-ции `ObjectStreamConstants.PROTO-COL_VERSION_1` (использовалась в JDK версии 1.1) или `ObjectStream-Constants.PROTOCOL_VERSION_2` (ис-пользуется по умолчанию начиная с версии JDK 1.2).
`void defaultWriteObject()`	Вызывается из метода `writeObject` сериализуемого класса для сохра-нения нестатических и нетранзитивых полей этого класса
`writeBoolean, writeByte,`	Методы, аналогичные методам класса

writeShort, writeChar, writeInt, writeLong, writeFloat, writeDOuble, writeUTF	DataOutputStream для записи в поток примитивных типов.

Класс java.io.ObjectInputStream

Предназначен для получения из потока ввода примитивных типов, подобно классу DataOutputStream и объектов (иерархически), которые были предварительно записаны с помощью класса ObjectOutputStream.

Конструктор класса ObjectInputStream:

ObjectInputStream(InputStream i)	Создает объект класса, связанный с входным потоком i.

Методы класса ObjectInputStream:

Object readObject()	Получает из потока заданный объект и восстанавливает его иерархически.
void defaultReadObject()	Вызывается из метода readObject сериализуемого клас-са для восстановления нестатических и нетранзитивых полей этого класса.
readBoolean, readByte, readShort, readChar, readInt, readLong, readFloat, readDouble, readUTF	Методы, аналогичные методам класса DataInputStream для чтения из потока примитивных типов.

В случае, если стандартного поведения для сериализации объекта недостаточно, можно определить в сериализуемом классе методы private void writeObject(ObjectOutputStream oos) и private void readObject(ObjectInputStream ois), и определить в них необходимые действия по сериализации.

Пример сериализации и восстановления объекта:

```
ObjectOutputStream oos = new ObjectOutputStream(
                          new FileOutputStream("file.ser"));
oos.writeObject(new Date());
oos.close();

ObjectInputStream ois = new ObjectInputStream(
                         new FileInputStream("file.ser"));
Date d = (Date) ois.readObject();
ois.close();
```

Интерфейс java.io.Externalizable

Предназначен для реализации классами, которым требуется нестандартное поведение при сериализации. В интерфейсе описаны 2 метода — `writeExternal(ObjectOutput o)` и `readExternal(ObjectInput I)`, предназначенный для записи и чтения состояния объекта. По умолчанию никакихе поля объекта, реализующего интерфейс `Externalizable`, в поток не передаются.

Контроль версий сериализуемого класса

Очевидно, что при сериализации объекта необходимо сохранять некоторую информацию о классе. Эта информация описывается классом `java.io.ObjectStreamClass`, в нее входит имя класса и идентификатор версии. Последний параметр важен, так как класс более ранней версии может не суметь восстановить сериализованный объект более поздней версии. Идентификатор класса хранится в переменной типа `long serialVersionUID`. В том случае, если класс не определяет эту переменную, то класс `ObjectOutputStream` автоматически вычисляет уникальный идентификатор версии для него с помощью алгоритма Secure Hash Algorithm (SHA). При изменении какой-либо переменной класса или какого-нибудь метода не-private происходит изменение этого значения. Для вычисления первоначального значения `serialVersionUID` используется утилита serialver.

Сериализация апплетов

Одной из областей применения сериализации объектов является сериализация апплетов. В тег `<APPLET>` добавляется аттрибут `OBJECT`, который используется вместо `CODE` для указания на файл сериализованного объекта. Встретив данный тег, программа-просмоторщик восстанавливает апплет из сериализованного объекта. Данный способ хорош тем, что позволяет использовать апплеты уже в инициализированном состоянии.

Апплеты можно сериализировать при помощи утилиты appletviewer. Для этого, после того как выполнятся методы `init()` и `start()`, необходимо выбрать в меню команду Stop, чтобы остановить выполнение апплета, и команду Save для сериализации апплета в файл. Если сериализация пройдет удачно, будет записан файл с расширением .ser. После чего можно использовать следующий HTML-код для апплета:

```
<APPLET OBJECT = "MySerialApplet.ser" WIDTH=400 HEIGHT=200>
</APPLET>
```

RMI – вызов удаленных методов

RMI (Remote method invocation) – технология распределенного объектного взаимодействия, позволяющая объекту, расположен-ному на стороне клиента, вызывать методы объектов, расположенных на стороне сервера (удаленных объектов). Для программиста вызов удаленного метода осуществляется так же, как и локального.

Структура RMI

Общая структурная организация технологии RMI приведена на рис. 8.

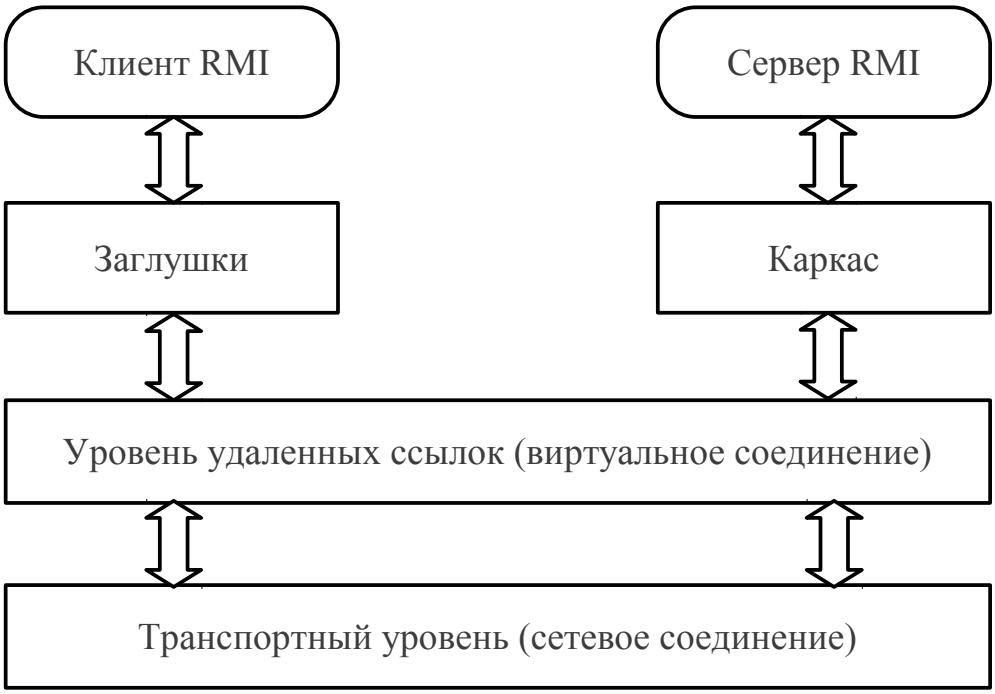

Рисунок 8. Структура RMI

Определения

Удаленный объект — объект, методы которого могут быть вызваны из другой виртуальной Java-машины, возможно расположенной на другой вычислительной системе.

Удаленный интерфейс — интерфейс, который реализуют удаленные объекты.

Вызов удаленного метода — действие по вызову метода и удаленного интерфейса, реализованного в удаленном объекте. Вызов такого метода имеет такой же синтаксис, как и вызов локального.

Сервер объектов — программа, предоставляющая удаленные методы для вызова.

Клиент — программа, осуществляющая вызов удаленных методов.

Каталог удаленных объектов (RMI Registry) — служебная программа, работающая на той же вычислительной системе, что и сервер объектов. Позволяет определять объекты, доступные для удаленных вызовов с данного сервера.

Объект-заглушка (Stub) - посредник удаленного объекта со стороны клиента. Предназначен для обработки аргументов и вызова транспортного уровня.

Алгоритм работы с RMI

1. Определение удаленных интерфейсов.
2. Создание сервера.
3. Создание клиента.
4. Запуск каталога удаленных объектов, сервера и клиента.

Определение удаленных интерфейсов

Требования к удаленному интерфейсу:

- должен иметь модификатор `public`;
- должен наследоваться от `java.rmi.Remote`;
- каждый метод интерфейса должен объявлять, что он выбрасывает `java.rmi.RemoteException`;
- аргументы и значения методов должны иметь примитивный или сериализуемый тип, либо тип удаленного интерфейса.

```
package hello;
import java.rmi.*;

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

Создание сервера

1. Объявление класса, реализующего удаленный интерфейс:

```
package hello;

public class Server implements Hello {

    public Server() { }

    public String sayHello() { return "Hello, World!"; }

}
```

2. Создание и экспорт удаленного объекта. Метод `exportObject` класса `java.rmi.server.UnicastRemoteObject` экпортирует удаленный объект, позволяя ему принимать удаленные вызовы на анонимный порт. Метод возвращает объект-заглушку, которая передается клиентам. Можно задать определенный порт, указав его в качестве второго аргумента. До версии JDK 1.5 заглушки создавались с помощью инструмента `rmic`. Этот способ необходимо применять в случае необходимости обеспечить совместимость с предущими версиями.

```
Server obj = new Server();

Hello stub = (Hello) UnicastRemoteObject.exportObject(obj,0);
```

3. Зарегистрировать удаленный объект в каталоге RMI registry:

```
Registry reg = LocateRegistry.getRegistry();

reg.bind("Hello", stub);
```

Создание клиентов

1. Получение ссылки на удаленный метод из каталога RMI registry:

```
Registry reg = LocateRegistry.getRegistry(hostname);
```

```
Hello stub = (Hello) registry.lookup("Hello");
```

2. Вызов удаленного метода

```
String response = stub.sayHello();

System.out.println(response);
```

Запуск каталога, сервера и клиентов

Для UNIX/Linux:

```
rmiregistry &

java -classpath path \
     -Djava.rmi.server.codebase=file:path/ Server &

java -classpath path Client
```

Для Windows:

```
start rmiregistry

start java -classpath path
-Djava.rmi.server.codebase=file:path/ Server

java -classpath path Client
```

Графический интерфейс пользователя

В Java существует две реализации графического пользовательского интерфейса — `java.awt` и `javax.swing`.

AWT — Abstract Window Toolkit

> `java.awt` - набор классов-оберток компонентов GUI операционной системы, на которой выполняется Java-приложение.

Компоненты AWT реализованы платформозависимым способом на каждой реализации java машины. При этом для пользователя существует иерархия оберток, обладающих разной реализацией графического представления на разных архитектурах.

Структура пакета `java.awt` приведена на рис. 9.

Компоненты

> *Компонент* (`java.awt.Component`) — базовый класс, определяющий отображение на экране и поведение каждого элемента интерфейса при взаимодействии с пользователем. Задает размер, цвет, отступы, область отображения и порождает основные события. Класс лежат в основе иерархии.

Для компонентов можно установить множество разных атрибутов, таких как цвет, шрифт и множество других.

Например, для установки цвета фона и переднего плана используются следующие методы:

```
setBackground(Color); // установка цвета фона компонент
setForeground(Color); // установка цвета на переднем плане
                      // (например, надпись на кнопке)
```

Для определения цвета используется класс `java.awt.Color`, в котором цвет можно задать либо при помощи таблицы RGB, либо при помощи переменных класса.

Для установки шрифта используется метод `setFont(Font)`. Все надписи, принадлежащие данному компоненту будут написаны эти шрифтом. Для определения шрифта используется класс `java.awt.Font`, в котором шрифт определяется через 3 параметра: имя (Helvetica), стиль его написания (BOLD) и высота (12).

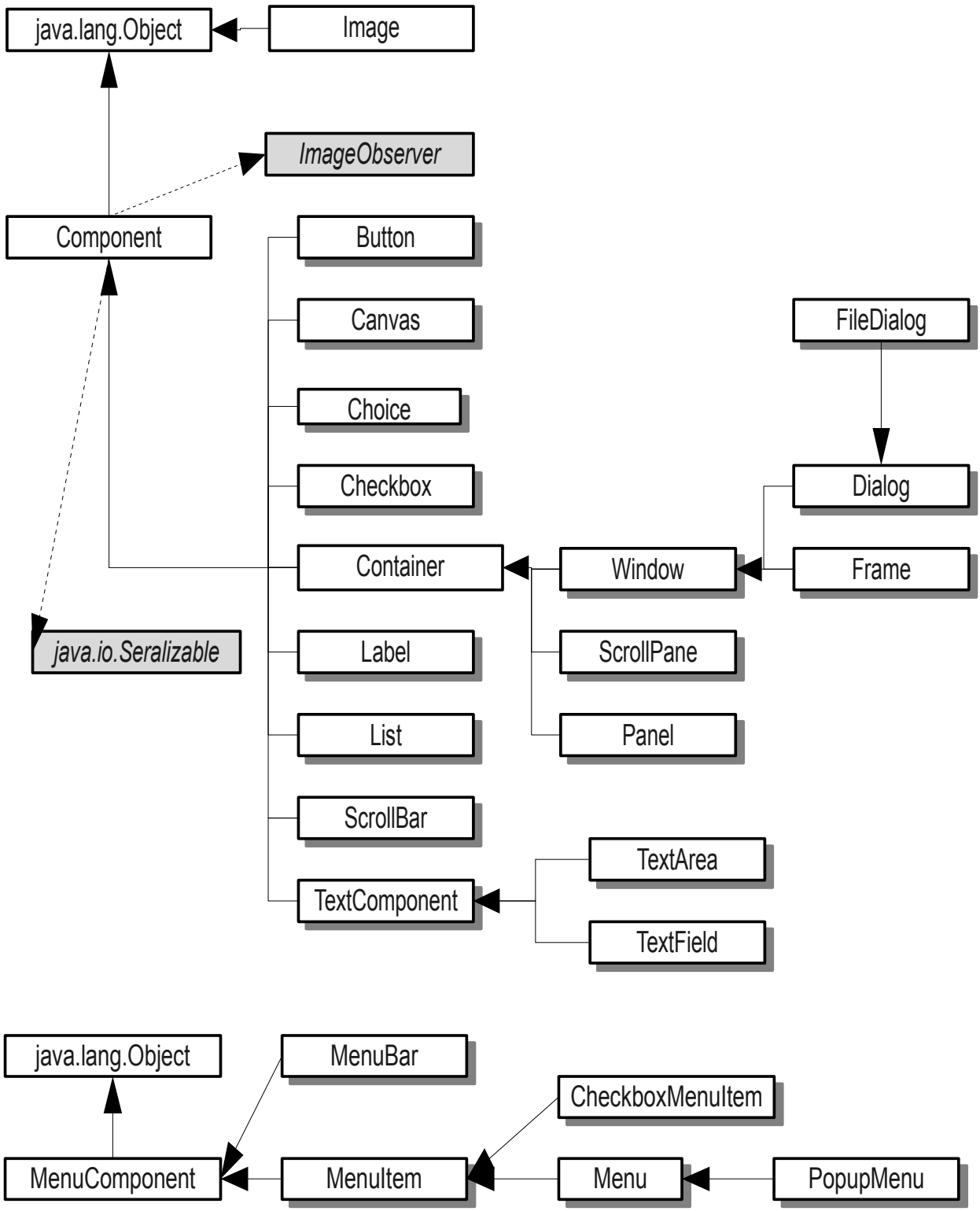

Рисунок 9. Структура пакета `java.awt`

Возможность включать изображения в программу реализуется через потомков абстрактного класса `Image`.

Кнопка (Button)

Внешний вид компонента приведён на рис. 10.

Рисунок 10. Кнопка

```
Button b = new  Button ("New Button");
```

Простейший компонент пользовательского интерфейса, применяемый практически во всех приложениях.

Для создания нового компонента необходимо создать объект этого класса, в приведенном примере в качестве аргумента передается строка-заголовок. При создании новой кнопки можно использовать конструктор без аргументов.

Переключатели (Checkbox)

Внешний вид компонента приведён на рис. 11.

Рисунок 11. Переключатели

```
Checkbox ch = new Checkbox ("One", true);
```

Данный компонент позволяет визуально устанавливать значения `on` или `off` для переменной. При использовании нескольких переключателей, несколько переменных могут принимать значение `on`.

При создании нового объекта необходимо задать как строку название, так и его значение. Необходимо помнить о том, что при использовании в качестве параметра объекта типа `CheckboxGroup` по крайней мере хотя бы один `CheckBox` должен быть в состоянии `on`.

Группы переключателей (CheckboxGroup)

Внешний вид компонента приведён на рис. 12.

Рисунок 12. Группа переключателей

```
CheckboxGroup cbg = new CheckboxGroup();
     add(new Checkbox("yellow", cbg, false));
     add(new Checkbox("green", cbg, true));
     add(new Checkbox("blue", cbg, false));
     add(new Checkbox("red", cbg, false));
```

Для использования этого компонента необходимо наличие n-числа переключателей. Алгоритм создания группы переключателей следующий: инициализация нового объекта типа `CheckboxGroup` и добавление к нему переключателей при помощи метода `add()`.

Списки (List)

> Списком называется набор элементов, один или несколько из которых могут быть выбраны из создаваемого окна с прокруткой.

Для создания списка необходимо:

- инициализация объекта типа `List` (в конструкторе задаем число видимых строк и аттрибут, определяющий возможен ли выбор нескольких строк одновременно);
- добавление строк методом `add()`.

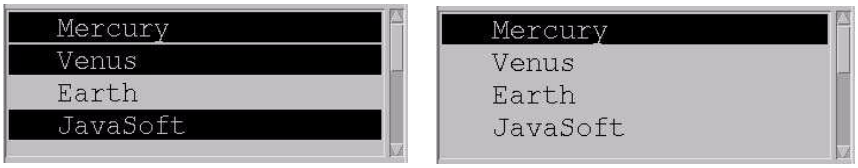

Рисунок 13. Группа переключателей

Возможно использование пустого конструктора. Примеры списков с использованием одиночного и множественного выбора представлены на рис. 13.

```
List lst = new List (4, false);
         lst.add("Mercury");
         lst.add("Venus");
         lst.add("Earth");
         lst.add("JavaSoft");
         lst.add("Mars");
```

```
        lst.add("Jupiter");
        lst.add("Saturn");
        lst.add("Uranus");
        lst.add("Neptune");
        lst.add("Pluto");
        lst.setMultipleMode (true);
```

Метод `setMultipleMode(boolean)` определяет возможность выбора множества строк.

Выпадающие списки (Choice)

```
Choice ColorChooser = new Choice();
    ColorChooser.add("Green");
    ColorChooser.add("Red");
     ColorChooser.add("Blue");
```

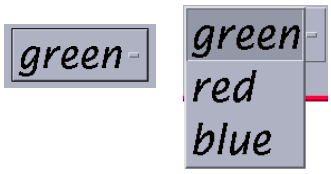

Рисунок 14. Выпадающие списки

Данный компонент (рис. 14)создается аналогично обычному списку, только при его использовании можно всегда выбирать только одну строку.

В данном классе используется только один пустой конструктор, который и создает объект этого класса, для полноценной работы которого необходимо добавление n-числа строк, что реализуется при помощи метода `add()`.

Полоса прокрутки (Scrollbar)

```
Scrollbar sb = new Scrollbar(Scrollbar.VERTICAL,
                             0, 1, 0, 255);
```

Рисунок 15. Полоса прокрутки

Как правило, полоса (рис. 15) автоматически добавляется к полю редактирования, но их можно использовать и самостоятельно, как отдельные элементы интерфейса. При создании нового объекта можно задать следующие параметры:

- тип - вертикальная или горизонтальная;
- начальное значение (положение);

- размер скроллера;
- минимальное значение;
- максимальное значение.

Надписи (Label)

```
Label l = new Label ("This is new Label", Label.CENTER);
    l.setText  ("This is Label");
    l.setAlignment  (Label.LEFT)
```

Рисунок 16. Надпись

Данный компонент (рис. 16) позволяет размещать статические текстовые надписи. При создании объекта можно использовать пустой конструктор, но тогда саму надпись необходимо задавать при помощи метода `setText(String)`. В приведенном примере задается строка надписи и тип выравнивания (`LEFT, RIGHT, CENTER`), последнее можно задать при помощи метода `setAlignment(int)`.

Холст (Canvas)

Данный класс предоставляет возможность выводить на экран изображения, а также при помощи графических примитивов создавать собственные. Делается это при использовании методов класса `Graphics` и метода `paint()`.

Текстовые Компоненты (TextComponent)

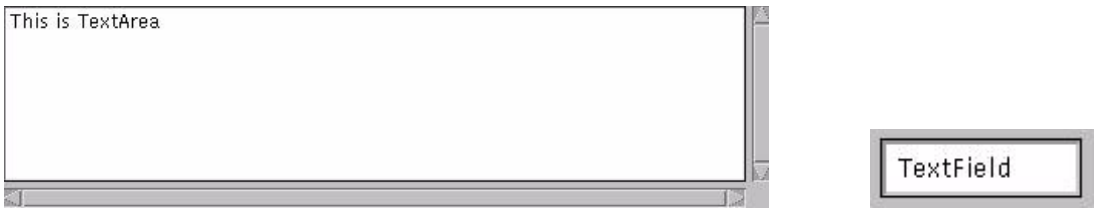

Рисунок 17. Текстовые компоненты

```
TextField tf = new TextField ();
    tf.setText ("This is TextField");
    tf.setColumn (10);

TextArea ta = new TextArea ("This is new TextArea", 10, 10,
TextArea. SCROLLBARS_BOTH);
```

Их можно разделить на два основных элемента (рис. 17) - строковое поле (`TextField`), текстовое поле (`TextArea`). Их отличие состоит в следующем: в текстовом поле может содержаться n-число строк и k-число столбцов, а строковое одну строку и k-число столбцов. Методы для этих полей сходны, только для строкового поля их меньше.

Для создании объекта типа `TextArea` возможно использование 5 конструкторов, рассмотрим один из них, который требует наибольшее количество параметров:

- строковая переменная, определяющая содержимое текстового поля;
- целочисленная переменная, определяющая количество строк в поле;
- целочисленная переменная, определяющая количество столбцов в поле;
- целочисленная переменная, определяющая тип визуализации горизонтальных и вертикальних полос прокрутки.
- В приведенном примере создается объект с заданным текстом, количество строк и столбцов равно 10 и с наличием вертикальных и горизонтальных полос прокрутки.

Контейнеры

> *Контейнер* (*`java.awt.Container`*) - компонент, способный содержать в себе другие компоненты, и управлять их размещением и, возможно, размерами при помощи *менеджеров компоновки*.

Панель (Panel)

Работа с данным компонентом не отличается от работы с другими компонентами класса `java.awt`, за единственным исключением, что для панели можно установить менеджер компоновки (см. далее) при помощи метода `setLayout()`. Добавление компонентов осуществляется методом `add()`.

В отличии от других компонентов панель не прорисовывается, то есть не имеет внешнего вида.

```
Panel p1 = new Panel ();
    p1.setLayout (new FlowLayout ());
    p1.add (new Button ("Button1"));
    p1.add (new TextField (10));
    p1.add (new Label ("This is FlowLayout"));
```

Скроллирующие панели (ScrollPane)

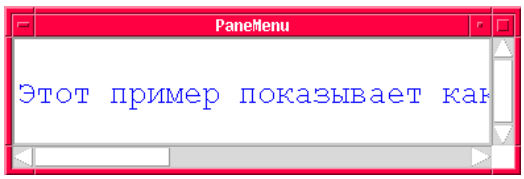

Рисунок 18. ScrollPane

Контейнер `ScrollPane` (рис. 18) отображает ограниченный участок дочернего элемента и снабжен горизонтальными и вертикальными полосами прокрутки для его просмотра в окне просмотра. Использование `ScrollPane` просто. Необходимо его создать и добавить дочерний элемент. `ScrollPane` отличается от обычного контейнера следующим:

- он поддерживает лишь один дочерний элемент (как правило, используется панель со множеством компонент);
- не предусмотрено использование менеджера компоновки
- необходимо задавать размеры с помощью метода setSize()

Окно приложения (Frame)

> Frame — контейнер, формирующий окна приложения с заголовком окна. Недоступен в апплетах.

Фрейм позволяет добавлять к себе меню и обрабатывать события интерфейса пользователя, связанные с активированием, сворачиванием и закрытием окна.

Менеджеры компоновки

Менеджер компоновки является незаменимым инструментом при использовании в апплете более одного компонента. Пакет `java.awt` включает следующие менеджеры:

- `FlowLayout` - менеджер, используемый по умолчанию, размещает компоненты последовательно в строку. Его использование оправдано в том случае, когда Вы знаете точные размеры компонент.
- `BorderLayout` — создает так называемое полярное расположение: разбивает панель на 5 зон (South, North, Center, West, East). Он учитывает разницу в размерах отдельных компонентов и пытается максимально использовать пространство контейнеров.
- `GridLayout` - создает решетку, состоящую из прямоугольников одинакового размера, в каждом из которых располагается один компонент.

- `CardLayout` - предназначен для последовательной визуализации различных панелей на одной основный.
- `GridBagLayout` - данный менеджер является наиболее сложным. Он позволят реализовывать сложный интерфейс, в котором контейнер содержит много компонентов различных размеров, которые должны находиться в одном и том же заданном положении относительно других.

В приведенном ниже примере рассматриваются все описанные выше компоненты.

```
import java.awt.*;
public class SimpleComponents extends java.applet.Applet {
public void init () {

     Panel p1 = new Panel ();
     Panel p2 = new Panel ();
     p1.setLayout (new FlowLayout ());
     p1.add (new Button ("Button1"));
     p1.add (new TextField (10));
     p1.add (new Label ("This is FlowLayout"));
     p2.setLayout (new GridLayout (3, 1, 0, 0));
     p2.add (new TextArea ("This is GridLayout", 10, 5));
     Choice ColorChooser = new Choice();
     ColorChooser.add("Green");
     ColorChooser.add("Red");
      ColorChooser.add("Blue");
      p2.add (ColorChooser);
     List lst = new List(4, false);
      lst.add("Mercury");
          lst.add("Venus");
          lst.add("Earth");
          lst.add("JavaSoft");
          lst.add("Mars");
          lst.add("Jupiter");
          lst.add("Saturn");
          lst.add("Uranus");
          lst.add("Neptune");
          lst.add("Pluto");
          lst.setMultipleMode (true);
          p2.add(lst);
          setLayout (new BorderLayout ());
          add ("South", p1);
          add ("Center", p2);
      }
}
```

Меню (Menu)

Меню - компонент пользовательского интерфейса, позволяющий создавать в приложениях главное меню. Меню в `java.awt` неразрывно связано с содержащим ее фреймом.

Процесс подсоединения меню к приложению (рис. 19) состоит из нескольких частей:

- создание строки меню

```
MenuBar mb = new MenuBar();
```

- создание нового меню на строку меню

```
Menu m = new Menu("File");
```

- добавление опции в меню и присваивание ей имени

```
m.add(new MenuItem ("Load")
```

- Добавление меню к фрейму

```
Frame f = new Frame("окно");
f.setMenuBar(mb);
```

Рисунок 19. Создание меню

Пример использования Меню

```
import java.awt.*;

public class SimpleMenu extends Frame {
MenuBar  mb;
Menu     file, edit, exit, help, subhelp;
MenuItem fload, fsave, fnew, ecopy, epaste, ecut,
         habout, hdetails;
CheckboxMenuItem eview;

public SimpleMenu () {
    mb = new MenuBar ();
```

```
    setMenuBar (mb);
    file = new Menu ("File", true);edit = new Menu ("Edit");
    exit = new Menu ("Exit");help = new Menu ("Help");
    fload = new MenuItem ("Load");
    fsave = new MenuItem ("Save");
    fnew = new MenuItem ("New");
    file.add (fload);
    file.add (fsave);
    file.add (fnew);
    ecopy = new MenuItem ("Copy");
    epaste = new MenuItem ("Paste");
    ecut = new MenuItem ("Cut");
    eview = new CheckboxMenuItem ("View");
    edit.add (ecopy);
    edit.add (epaste);
    edit.add (eview);
    subhelp = new Menu ("Sub Help Menu");
    habout = new MenuItem ("About");
    hdetails = new MenuItem ("Details");
    mb.setHelpMenu (help);
    help.add (subhelp);
    subhelp.add (habout);
    subhelp.add (hdetails);
    mb.add (file);
    mb.add (edit);
    mb.add (exit);
    mb.add (help);
    setSize (200, 300);
    }

public static void main (String args []) {
    SimpleMenu sm = new SimpleMenu ();
    sm.pack ();
    sm.setVisible (true);
    }
}
```

Модель обработки событий

Компоненты AWT генерируют события в соответствии с воздействиями пользователя на графический интерфейс. Другие компоненты регистрируются для прослушивания этих событий и реагируют соответствующим образом.

Существует 4 составные части модели обработки событий.

- Компонент-источник события
- Компоненты-слушатели или приемники события
- Интерфейс слушателя
- Класс события

Источник события

> *Источником события* является компонент, генерирующий событие и регистрирующий заинтересованные в прослушивании данного события компоненты.

К примеру, реализация источника события нажатия на кнопку для класса `Button` могла бы быть выполнена следующим образом:

```
public class Button extends Component {

private String label = null; // надпись на кнопке
private Vector listeners = new Vector(); //слушатели события
public synchronized void addActonListener(ActonListener l) {
    listeners.addElement (l); // Регистрация слушателя
}
public synchronized void removeBeginListener(ActonListener l)
{
    listeners.removeElement (l); // Удаления слушателя
}

//оповещение слушателей (например, при щелчке мышки на
кнопке)
protected void notifyActon () {
    Vector copy;
    //создание события;
    ActionEvent h = new ActionEvent(this,0,label);
    synchronized (this) {copy = (Vector)listeners.clone();}
    for (int i=0;i<l.size();i++) {
        ActonListener bl = (ActonListener)l.elementAt (i);
        bl.actionPerformed(h);
    }
```

```
} }
```

Источник события оповещает слушателя путем вызова специального полиморфного метода интерфейса слушателя (`actionPerformed`) и передачи ему объекта события (`ActionEvent`). Все слушатели событий определенного события должны реализовывать соответствующий интерфейс.

Внимание! Реальная реализация Button в AWT использует более эффективный код, использующий `java.awt.AWTEventMulticaster`.

Интерфейс слушателя

> *Интерфейс слушателя* — вспомогательный интерфейс, который обязаны реализовывать все слушатели события для возможности добавления себя в список у источника события.

```
package java.awt.event;

public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}
```

Иинтерфейс определяет методы (здесь `actionPerformed`), которые будут вызваны в момент доставки события слушателю. Все интерфейсы слушателей являются расширением класса `java.util.EventListener`. По установленному соглашению, методам слушателей может быть передан один единственный аргумент, который соответствует данному слушателю.

Слушатель события

> *Слушатель события* — компонент, регистрирующийся для прослушивания события у источника и реагирующий на него.

Слушатель может события обрабатывать внутри собственного класса:

```
public class MyFrame extends Frame implements ActionListener
{
    Button b = new Button("New"); // кнопка источник-события

    public MyFrame () {
        b.addActionListener(this); //регистрация у источника
    }

    // реализация интерфейса слушателя
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button "+b+" was pressed");
```

```
    }
}
```

Слушатель может использовать неименованные (анонимные) классы для прослушивания события. Это упрощает код слушателя и позволяет реализовать в одном классе много слушателей однотипных событий:

```
// кнопка источник-события и регистрация у источника
Button b = new Button("New");
b.addActionListener(new ActionListener(){
    // реализация интерфейса слушателя
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button "+e.getActionCommand()+
                           " was pressed");
    }
});
```

Здесь происходит неявное реализация интерфейса `ActionListener` новым, анонимным классом, с единственным методом `actionPerformed`.

Событие

> *Событие пользовательского интерфейса* — потомок класса `java.awt.AWTEvent`, предназначенный для передачи информации от источника события к слушателю.

События пакета AWT вместе со слушателями находятся в пакете `java.awt.event`.

В событии должна содержаться вся информация, необходимая программе для формирования реакции на данное событие. В нашем примере это аттрибут `actionCommand`.

```
package java.awt.event;

public class ActionEvent extends AWTEvent {

    String actionCommand;

    public ActionEvent(Object source,int id,String command) {
        super(source, id);
        this.actionCommand = command;
    }
    public String getActionCommand() { return actionCommand;}
}
```

Класс AWTEventMulticaster

AWTEventMulticaster — класс, реализующий эффективную и потокобезопасную диспетчеризацию событий для событий AWT.

Характерное использование класса:

```
public NewAWTComponent extends Component {
    ActionListener al = null;

public synchronized void
addActionListener(ActionListener l) {
     al = AWTEventMulticaster.add(al, l);
}
public synchronized void
removeActionListener(ActionListener l) {
     al = AWTEventMulticaster.remove(al, l);
}
public void processEvent(AWTEvent e) {
     ActionListener l = al;
     if (l != null) {
        l.actionPerformed(new ActionEvent(this, 0,
"action"));
     }
}
}
```

В Swing для подобного функционала используется `EventListenerList`.

Основные события AWT

Ссобытие, Интерфейс слушателя	Элемент	Методы слушателя	Значение
`ActionEvent, ActionListener`	`Button`	`actionPerformed`	нажатие кнопки
	`List`		двойной щелчок
	`MenuItem`		выбор пункта меню
	`TextField`		конец редактирования текста элемента, ENTER
`AdjustmentEvent AdjustmentListener`	`Scrollbar`	`adjustmentValueChanged`	реализация прокрутки
`ComponentEvent ComponentListen`	`Component`	`componentHidden componentMoved`	перемещение, изменение размеров, стал скрытым или

er		componentResized componentShown	видимым
ContainerEvent ContainerListener	Container	componentAdded componentRemoved	добавление или удаление элемента в контейнер
FocusEvent FocusListener	Component	focusGained focusLost	получение или потеря фокуса
ItemEvent ItemListener	Checkbox	itemStateChanged	установка (сброс) флажка
	CheckboxMenuItem		установка (сброс) флажка у пункта меню
	Choice		выбор элемента списка
	List		выбор элемента списка
KeyEvent, KeyListener	Component	keyPressed keyReleased keyTyped	нажатие или отпускание клавиши
MouseEvent, MouseListener MouseMotionListener	Component	mouseClocked mouseEntered mouseExited mousePressed mouseReleased mouseDragged mouseMoved	нажатие или отпускание кнопки мыши, курсор мыши вошел или покинул область элемента, перемещение мыши, перемещение мыши при нажатой кнопки мыши.
TextEvent, TextListener	TextComponent	textValueChahged	изменения в тексте элемента
WindowEvent, WindowListener	Window	windowActivated windowClosed windowClosing windowDeactivated windowDeiconified windowOpened	открытие, закрытие, минимизация, восстановление, запрос на обновление окна

JFC - Java Foundation Classes

JFC — набор базовых библиотек, предназначенный для построения эффективных графических приложений.

Состоит из:

- AWT - Содержит компоненты AWT 1.1
- Java 2D - Обеспечивает расширенные возможности создания и обработки изображений.
- Accessibility - Реализует вспомогательные технологии, такие как экранное считывание, обработка речи, и т.д.
- Drag and Drop - Дает возможность взаимодействия приложений на языке Java с другими приложениями с использованием механизма Drag and Drop.
- Swing - расширяет набор компонентов AWT 1.1 путем добавления легковесных компонентов, реализующих архитектуру MVC, в отличие от компонентов, основанных на использовании экспертов операционной системы.

Swing

Набор компонентов Swing - это разработанный на Java платформо-независимый набор компонентов графического пользовательского интерфейса.

Особенности Swing-компонентов:

- являются "легковесными", т.е. не используют средств операционной системы.
- реализуют архитектуру PL&F (Pluggable Look and Feel), позволяя легко изменять вид и поведение работающего приложения.
- реализуют компонентную технологию JavaBeans, и могут использоваться с любыми визуальными средствами разработки, поддерживающими работу с Beans-компонентами.

Архитектура MVC и модель Swing

MVC (Model-View-Controller) - технология создания элементов пользовательского интерфейса (рис. 20), основанная на взаимодействии 3 компонентов:

- Model (модель) - логическое представление данных
- View (представление) - визуальное представление данных
- Controller (контроллер) - обрабатывает входные данные и передает их изменения в модель.

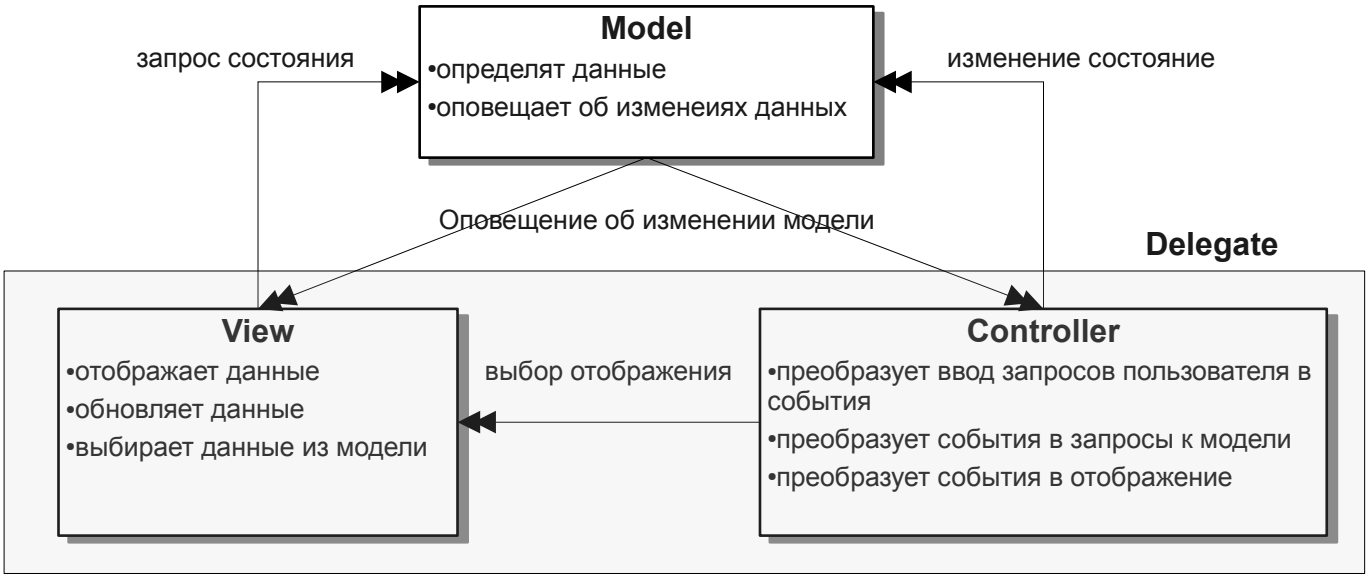

Рисунок 20. Модель MVC

В Swing контроллер и представление объединяются в общий компонент под названием *делегат (delegate)*.

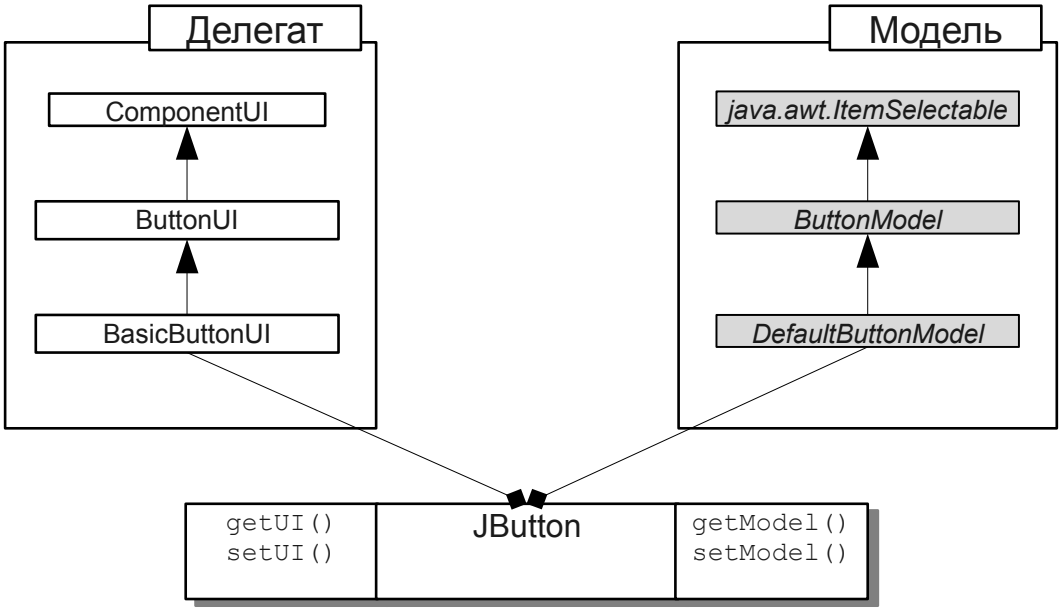

Рисунок 21. Модели и делегаты.

Компоненты Swing в любой момент времени связаны с определенной моделью и с определенным делегатом (рис. 21), специфическими для данного компонента. Модели и делегаты реализуют соответствующие интерфейсы.

Внешний вид компонент (Look & Feel)

Так как в архитектуре MVC используется модульный принцип построения компонентов, то легко изменить внешний вид всех компонентов одновременно. Существует несколько стандартных и множество сторонних реализаций внешнего вида компонентов.

```
//Устанавливает L&F, зависящий от платформы,
//на которой работает программа
try {
  UIManager.setLookAndFeel
    (UIManager.getSystemLookAndFeelClassName());
} catch (java.lang.ClassNotFoundException e) {}

//Устанавливает Java L&F
try {
  UIManager.setLookAndFeel
    (UIManager.getCrossPlatformLookAndFeelClassName());
} catch (java.lang.ClassNotFoundException e) {}
```

Для изменения вида уже созданных и отображенных компонентов используется метод `updateUI()` каждого компонента. Для облегчения этого процесса можно вызвать метод `SwingUtilities.updateComponentTreeUI()`.

Пакеты Swing

javax.swing	Содержит платформонезависимую реализацию компонентов, адаптеров, моделей компонент и интерфейсов для делегатов и моделей
javax.swing.border	Объявляет интерфейс Border и реализующие его классы, для отображения бордюров - визуального обрамления компонентов Swing
javax.swing.colorchooser	Реализует компонент JColorChooser
javax.swing.event	В дополнение к стандартным типам событий, компоненты Swing могут порождать собственные события, собранные в этом пакете
javax.swing.filechooser	Реализует компонента JFileChooser
javax.swing.plaf	Содержит интерфесы и множество абстрактных

	классов для поддержки PLAF (Pluggable Look-and-Feel) компонентов Swing
javax.swing.plaf.basic	Содержит классы (делегаты) интерфейса пользователя, которые реализуют базовое (кодовое имя - baisc) отображение компонентов Swing
javax.swing.plaf.metal	Реализует принятое по умолчанию Java-отображение компонентов Swing (кодовое имя - metal)
javax.swing.plaf.multi	Позволяет использовать несколько отображений компонентов Swing в одном приложении.
javax.swing.plaf.synth	Содержит классы и интерфейсы для реализации собственного отображения компонент.
javax.swing.table	Содержит интерфейсы и классы для поддержки работы с таблицами
javax.swing.text	Содержит классы поддержки работы с текстовыми компонентами
javax.swing.text.*	Содержит классы обработки текста в формате HTML и RTF
javax.swing.tree	Включает компоненты для JTree
javax.swing.undo	Обеспечивает поддержку undo/redo.

Составные части окна

В модели Swing внутренняя часть окна представляет собой корневую панель `JRootPane`, которая в свою очередь состоит из прозрачной панели (*glass pane*) и слоистой панели (*layered pane*). Прозрачная панель невидима, и используется для отображения подсказок и выплывающих меню. Слоистая панель тоже состоит из двух компонентов - линейки меню и панели содержимого (*content pane)*. Компоновка осуществляется на панели содержимого:

```
aFrame.getContentPane().setLayout (new FlowLayout());
aFrame.getContentPane().add(aComponent);
```

При помещении компонента на слоистую панель можно указывать номер слоя, на который следует поместить компонент:

```
layeredPane.add (component, new Integer(5));
```

Класс JComponent

Является базовым (рис. 22) классом почти для всех Swing-компонентов пользовательского интерфейса (J-классов). Swing-компоненты

пользовательского интерфейса наследуют от класса JComponent следующие свойства:

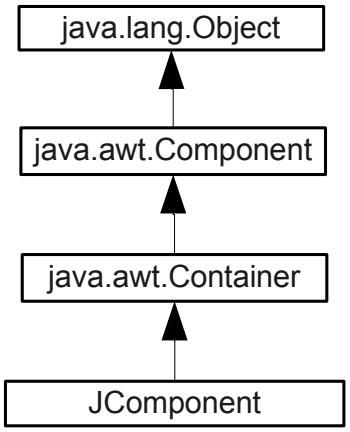

Рисунок 22. JComponent

- реализация PL&F
- расширяемость
- обработка событий клавиатуры
- прорисовка границ
- масштабируемость
- отображение подсказок
- автопрокрутка
- возможность отладки графики
- поддержка вспомогательных технологий
- многоязыковая поддержка

Класс JPanel

Является легковесным объектом, представляющим панель со встроенной поддержкой двойной буферизации.

Интерфейс Icon и класс ImageIcon

Интерфейс `Icon` описывает изображения фиксированного размера. Класс `ImageIcon` реализует интерфейс `Icon` для объекта типа `java.awt.Image`.

```
public interface Icon {
  void paintIcon(Component c, Graphics g, int x, int y);
  int getIconWidth();
  int getIconHeight();
}
```

Отличия `ImageIcon` от `java.awt.Image`:

- не требует монитора при загрузке
- является сериализуемым

Если объект типа `IconImage` использует динамически меняющееся изображение, то для его отображения необходимо связать наблюдатель `ImageObserver` с изображением `IconImage`. Обычно наблюдателем является тот компонент, на котором изображается `ImageIcon`.

```
IconImage picture = new IconImage("pic.gif");
Jbutton button = new Jbutton(picture);
picture.setImageObserver(button);
```

Класс JLabel

Реализует однострочную текстовую метку с дополнительными возможностями по сравнению с `java.awt.Label`, такими как:

- Наличие изображения
- Возможность изменения взаимного расположения текста и изображения

Пример использования:

```
public class LabelPanel extends JPanel {
  public LabelPanel() {
    // Создать и добавить JLabel
    JLabel plainLabel = new JLabel("Plain Small Label");
    add(plainLabel);
    // Создать вторую JLabel
    JLabel fancyLabel = new JLabel("Fancy Big Label");
    // Реализовать объект Font для использования на метке
    Font fancyFont = new Font("Serif",
                              Font.BOLD | Font.ITALIC, 32);
    // Связать шрифт и метку
    fancyLabel.setFont(fancyFont);
    // Создать изображение Icon
    Icon tigerIcon = new ImageIcon("SmallTiger.gif");
    // Поместить изображение на метку
    fancyLabel.setIcon(tigerIcon);
    // Поместить текст справа от изображения
    fancyLabel.setHorizontalAlignment(JLabel.RIGHT);
    // Поместить на панель
    add(fancyLabel);
  }
}
```

Класс JButton

Является реализацией кнопки с возможностью включения изображения. Фон кнопки должен совпадать с фоном контейнера, в котором она находится.

```
public class ButtonPanel extends JPanel {
  public ButtonPanel() {
    Icon tigerIcon = new ImageIcon("SmallTiger.gif");
    JButton myButton = new JButton("Tiger", tigerIcon);
    myButton.setBackground (SystemColor.control);
    add(myButton);
  }
}
```

Класс JCheckBox

Соответствует объекту CheckBox, не входящему в группу. Можно задавать собственные изображения для выбранного и невыбранного состояния.

```
public class CheckboxPanel extends JPanel {
  Icon unchecked = new ToggleIcon (false);
  Icon checked = new ToggleIcon (true);
  public CheckboxPanel() {
    // Установить компоновку для JPanel
    setLayout(new GridLayout(2, 1));
    // Создать элемент выбора с состоянием true
    JCheckBox cb1 = new JCheckBox("Choose Me", true);
    cb1.setIcon(unchecked);
    cb1.setSelectedIcon(checked);
    // Создать элемент выбора с состоянием false
    JCheckBox cb2 = new JCheckBox("No Choose Me", false);
    cb2.setIcon(unchecked);
    cb2.setSelectedIcon(checked);
    add(cb1);
    add(cb2);
  }
  class ToggleIcon implements Icon {
    boolean state;
    public ToggleIcon (boolean s) {
      state = s;
    }
    public void paintIcon (Component c, Graphics g,
                           int x, int y) {
      int width = getIconWidth();
      int height = getIconHeight();
      g.setColor (Color.black);
      if (state)
        g.fillOval (x, y, width, height);
      else
        g.drawOval (x, y, width, height);
    }
    public int getIconWidth() {
      return 10;
    }
    public int getIconHeight() {
      return 10;
    }
  }
}
```

Класс JRadioButton

Соответствует объекту `CheckBox`, входящему в группу, так, что в данный момент времени может быть выбран только один из них. Группа объектов задается с помощью `ButtonGroup`.

```
public class RadioButtonPanel extends JPanel {
  public RadioButtonPanel() {
    // Установить компоновку GridLayout
    setLayout(new GridLayout(4,1));
    // Объявление переключателя
    JRadioButton radioButton;
    // Реализация группы для переключателей
    ButtonGroup rbg = new ButtonGroup();
    // Создание метки для группы
    JLabel label = new JLabel("Annual Salary: ");
    label.setFont(new Font("SansSerif", Font.BOLD, 14));
    add(label);
    // Добавить новый переключатель
    radioButton = new JRadioButton("$45,000");
    add (radioButton);
    // установить "горячую" клавишу
    radioButton.setKeyAccelerator ('4');
    // Добавить переключатель в группу
    rbg.add (radioButton);
    // Установить данный переключатель по умолчанию
    radioButton.setSelected(true);
    // Установить еще 2 переключателя
    radioButton = new JRadioButton("$60,000");
    radioButton.setKeyAccelerator ('6');
    add (radioButton);
    rbg.add (radioButton);
    radioButton = new JRadioButton("$75,000");
    radioButton.setKeyAccelerator ('7');
    add (radioButton);
    rbg.add (radioButton);
  }
}
```

Класс JToggleButton

Является суперклассом для `JCheckBox` и `JRadioButton`. Кнопка остается в нажатом состоянии, и возвращается в исходное состояние при повторном нажатии.

```
public class ToggleButtonPanel extends JPanel {
  public ToggleButtonPanel() {
    // Установить компоновку GridLayout
```

```
    setLayout(new GridLayout(4,1, 10, 10));
    add (new JToggleButton ("Fe"));
    add (new JToggleButton ("Fi"));
    add (new JToggleButton ("Fo"));
    add (new JToggleButton ("Fum"));
  }
}
```

Класс JTextComponent

Класс обеспечивает основные функции простого текстового редактора. Основными подклассами `JTextComponent` являются `JTextField`, `JTextArea`, `JTextPane`.

```
JTextField tf = new JTextField();
JTextArea ta = new JTextArea();
tf.setText("TextField");
ta.setText("JTextArea\n Allows Multiple Lines");
add(tf);add(ta);
JTextPane tp = new JTextPane();
MutableAttributeSet attr = new SimpleAttributeSet();
StyleConstants.setFontFamily(attr, "Serif");
StyleConstants.setFontSize(attr, 18);
StyleConstants.setBold(attr, true);
tp.setCharacterAttributes(attr, false);
add(tp);
```

Класс JScrollBar

Реализует полосу прокрутки

```
public class ScrollbarPanel extends JPanel {
  public ScrollbarPanel() {
    setLayout(new BorderLayout());
    JScrollBar sb1 =
      new JScrollBar (JScrollBar.VERTICAL, 0, 5, 0, 100);
    add(sb1, BorderLayout.EAST);
    JScrollBar sb2 =
      new JScrollBar (JScrollBar.HORIZONTAL, 0, 5, 0, 100);
    add(sb2, BorderLayout.SOUTH);
  }
}
```

Класс JSlider

Реализует полосу прокрутки с возможностью разметки.

```
public class SliderPanel extends JPanel {
```

```
  public SliderPanel() {
    setBackground (Color.lightGray);
    setLayout(new BorderLayout());
    setBackground (Color.lightGray);
    JSlider s1 = new JSlider (JSlider.VERTICAL, 0, 100, 50);
    s1.setPaintTicks(true);
    s1.setMajorTickSpacing(10);
    s1.setMinorTickSpacing(2);
    add(s1, BorderLayout.EAST);
    JSlider s2 = new JSlider (JSlider.VERTICAL, 0, 100, 50);
    s2.setPaintTicks(true);
    s2.setMinorTickSpacing(5);
    add(s2, BorderLayout.WEST);
    JSlider s3 = new JSlider (JSlider.HORIZONTAL, 0, 100,
50);
    s3.setPaintTicks(true);
    s3.setMajorTickSpacing(10);
    add(s3, BorderLayout.SOUTH);
    JSlider s4 = new JSlider (JSlider.HORIZONTAL, 0, 100,
50);
    s4.setBorder(LineBorder.createBlackLineBorder());
    add(s4, BorderLayout.NORTH);
  }
}
```

Класс JProgressBar

Данный класс позволяет отображать линейку прогресса, показывая ход выполнения операции. Использование JProgressBar:

```
/Инициализация компонента
JProgressBar progressBar = new JProgressBar();
progressBar.setMinimum(0);
progressBar.setMaximum(numberSubOperations);
//Выполнение операции
progressBar.setValue(progressBar.getMinimum());
for (int i = 0; i < numberSubOperations; i++) {
  // Установка значения
  progressBar.setValue(i);
}
```

Класс JComboBox

Представляет собой выпадающий список с возможностью выбора и редактирования.

```
public class ComboPanel extends JPanel {
  String choices[] = {"Mercury", "Venus", "Earth", "Mars",
```

```
    "Saturn", "Jupiter", "Uranus","Neptune","Pluto"};
  public ComboPanel() {
    JComboBox combo1 = new JComboBox();
    JComboBox combo2 = new JComboBox();
    for (int i=0;i<choices.length;i++) {
      combo1.addItem (choices[i]);
      combo2.addItem (choices[i]);
    }
    combo2.setEditable(true);
    combo2.setSelectedItem("X");
    combo2.setMaximumRowCount(4);
    add(combo1);
    add(combo2);
  }
}
```

Класс JList

Реализует список элементов.

```
public class ListPanel extends JPanel {
  String label [] = {"Cranberry", "Orange", "Banana",
    "Kiwi", "Blueberry", "Pomegranate", "Apple", "Pear",
    "Watermelon", "Raspberry", "Snozberry"
  };
  public ListPanel() {
    setLayout (new BorderLayout());
    JList list = new JList(label);
    ScrollPane pane = new ScrollPane();
    pane.add (list);
    add(pane, BorderLayout.CENTER);
  }
}
```

Класс JScrollPane

Реализует область прокрутки.

```
public class ScrollPanel extends JPanel {
  public ScrollPanel() {
    setLayout(new BorderLayout());
    Icon bigTiger = new ImageIcon("BigTiger.gif");
    JLabel tigerLabel = new JLabel(bigTiger);
    JScrollPane scrollPane = new JScrollPane();
    scrollPane.getViewport().add(tigerLabel);
    add(scrollPane, BorderLayout.CENTER);
```

```
  }
}
```

Бордюры (Border)

Рисование границ вокруг компонентов (бордюров) обеспечивается с помощью интерфейса `Border`. `Border` требует реализации следующих методов:

```
// Определяет область, необходимую для рисования
public Insets getBorderInsets(Component c);
// Определяет прозрачность и непрозрачность границы
public boolean isBorderOpaque();
// Определяет каким образом рисовать границу
public void paintBorder (Component c, Graphics g,
                         int x, int y,
                         int width, int height)
```

В пакет Swing входит 9 классов для рисования бордюров

`BevelBorder`	Объемный бордюр, поднятый или опущенный
`CompoundBorder`	Составной бордюр
`DefaultBorder`	Бордюр по умолчанию — реализация интерфейса
`EmptyBorder`	Пустой бордюр
`EtchedBorder`	Бордюр-канавка
`LineBorder`	Цветной бордюр произвольной толщины
`MatteBorder`	Бордюр с заполнением цветом или изображением
`SoftBevelBorder`	Объемный бордюр с закругленными углами
`TitledBorder`	Бордюр с заголовками в различных позициях

Классы меню

Классы, обеспечивающие работу с меню (`JCheckBoxMenuItem`, `JMenuItem`, `JRadioButtonMenuItem`, `JMenu`, `JMenuBar`, `JSeparator`), являются подклассами компонента `JComponent`. Это позволяет в отличие от AWT добавить меню в любое место программы, к любому контейнеру.

```
public class MenuTester extends JFrame implements
ActionListener {

  public void actionPerformed (ActionEvent e) {
    System.out.println (e.getActionCommand());
  }
```

```
  public MenuTester() {
    super ("Menu Example");
    JMenuBar jmb = new JMenuBar();
    JMenu file = new JMenu ("File");
    JMenuItem item;
    file.add (item = new JMenuItem ("New"));
    item.addActionListener (this);
    file.add (item = new JMenuItem ("Open"));
    item.addActionListener (this);
    file.addSeparator();
    file.add (item = new JMenuItem ("Close"));
    item.addActionListener (this);
    jmb.add (file);
    JMenu edit = new JMenu ("Edit");
    edit.add (item = new JMenuItem ("Copy"));
    item.addActionListener (this);
    edit.add (item = new JMenuItem ("Woods", tigerIcon));
    item.setHorizontalTextPosition (JMenuItem.RIGHT);
    item.addActionListener (this);
    JCheckBoxMenuItem check= new JCheckBoxMenuItem
("Toggle");
    check.addActionListener (this);
    edit.add (check);jmb.add (edit);
    setJMenuBar (jmb);
  }}
```

Классы JFrame, JWindow, JDialog

Основное отличие этих классов от других компонентов Swing заключается в том, что они не являются легковесными компонентами, и наследуются соответственно от `Frame`, `Window`, `Dialog`. В отличие от `Frame`, `Window`, `Dialog`, для добавления компонентов в `JFrame`, `JWindow`, `JDialog` используется панель содержимого (*content pane*).

```
public class FrameTester {
  public static void main (String args[]) {
  JFrame f = new JFrame ("JFrame Example");
  Container c = f.getContentPane();
  c.setLayout (new FlowLayout());
  for (int i = 0; i < 5; i++) {
    c.add(new
JButton("No")).setBackground(SystemColor.control);
    c.add (new Button("Batter"));
  }
  c.add (new JLabel ("Swing"));
  f.setSize (300, 200);
  f.show();
```

```
}
```

Класс JPopupMenu

Позволяет связать выпадающее меню с любым компонентом.

```
public class PopupPanel extends JPanel {
  JPopupMenu popup = new JPopupMenu ();
  public PopupPanel() {
    JMenuItem item;
    popup.add (item = new JMenuItem ("Cut"));
    popup.add (item = new JMenuItem ("Copy"));
    popup.add (item = new JMenuItem ("Paste"));
    popup.addSeparator();
    popup.add (item = new JMenuItem ("Select All"));
    enableEvents (AWTEvent.MOUSE_EVENT_MASK);
  }
  protected void processMouseEvent (MouseEvent e) {
    if (e.isPopupTrigger())
      popup.show (e.getComponent(), e.getX(), e.getY());
    super.processMouseEvent (e);
  }
}
```

Подсказки (ToolTip)

В пакете swing имеется класс `JtoolTip`, предназначенный для реализации подсказок, однако обычно достаточно вызвать метод `setToolTip()` класса `JComponent`.

```
public class TooltipPanel extends JPanel {
  public TooltipPanel() {
    JButton myButton = new JButton("Hello");
    myButton.setBackground (SystemColor.control);
    myButton.setToolTipText ("World");
    add(myButton);
  }
}
```

Класс JToolBar

Используется для реализации линейки инструментов.

```
public class ToolbarPanel extends JPanel {
  ToolbarPanel() {
    setLayout (new BorderLayout());
    JToolBar toolbar = new JToolBar();
```

```
    JButton myButton = new JButton("Hello");
    toolbar.add(myButton);
    Icon tigerIcon = new ImageIcon("SmallTiger.gif");
    myButton = new JButton(tigerIcon);
    toolbar.add(myButton);
    toolbar.addSeparator();
    toolbar.add (new Checkbox ("Not"));
    add (toolbar, BorderLayout.NORTH);
  }
}
```

Класс JTabbedPane

Реализует панель с закладками.

```
public class TabbedPanel extends JPanel {
  String tabs[] = {"One", "Two", "Three", "Four"};
  public JTabbedPane tabbedPane = new JTabbedPane();
  public TabbedPanel() {
    setLayout(new BorderLayout());
    for (int i=0;i<tabs.length;i++)
       tabbedPane.addTab (tabs[i], null, createPane
(tabs[i]));
    tabbedPane.setSelectedIndex(0);
    add (tabbedPane, BorderLayout.CENTER);
  }
  JPanel createPane(String s) {
    JPanel p = new JPanel();
    p.setBackground (SystemColor.control);
    p.add(new JLabel(s));
    return p;
  }
}
```

Класс JSplitPane

Позволяет упаковать 2 компонента с движком между ними.

```
public class JSplitPanel extends JPanel {
  JComponent createSplitter (int orientation, boolean depth)
{
    JButton butt1 = new JButton ("One");
    butt1.setBackground (SystemColor.control);
    JComponent c;
    if (depth) {
      int newOrientation =
        ((orientation == JSplitPane.HORIZONTAL_SPLIT) ?
         newOrientation = JSplitPane.VERTICAL_SPLIT :
```

```
          newOrientation = JSplitPane.HORIZONTAL_SPLIT);
      c = createSplitter (newOrientation, false);
    } else {
      c = new JButton ("Two");
      c.setBackground (SystemColor.control);
    }
    JSplitPane jsp = new JSplitPane (orientation, butt1, c);
    return jsp;
  }
  public JSplitPanel() {
    // Установить компоновку только для одного компонента
    setLayout(new BorderLayout(10, 10));
    add (createSplitter (JSplitPane.HORIZONTAL_SPLIT, false),
      BorderLayout.NORTH);
    add (createSplitter (JSplitPane.HORIZONTAL_SPLIT, false),
      BorderLayout.SOUTH);
    add (createSplitter (JSplitPane.VERTICAL_SPLIT, false),
      BorderLayout.WEST);
    add (createSplitter (JSplitPane.VERTICAL_SPLIT, false),
      BorderLayout.EAST);
    add (createSplitter (JSplitPane.VERTICAL_SPLIT, true),
      BorderLayout.CENTER);
  }
}
```

Класс BoxLayout

Менеджер компоновки для размещения компонентов вдоль оси x или y.

```
class BoxLayoutTest extends JPanel {
  BoxLayoutTest() {
    // Установить компоновку вдоль оси y
    setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
    TextField textField = new TextField();
    TextArea textArea = new TextArea(4, 20);
    JButton button = new JButton(
      "Tiger", new ImageIcon("SmallTiger.gif"));
    button.setBackground (SystemColor.control);
    add(new JLabel("TextField:"));
    add(textField);
    add(new JLabel("TextArea:"));
    add(textArea);
    add(new JLabel("Button:"));
    add(button);
  }
}
```

События пакетов Swing

AncestorEvent	предок изменен, перемещен, удален
ChangeEvent	изменение состояния
DocumentEvent	изменение состояния документа
DragEvent	событие drag-and-drop
ListDataEvent	изменение данных списка
ListSelectionEvent	изменение состояния выбора списка
MenuEvent	событие меню
TableColumnModelEvent	изменение модели столбца таблицы
TableModelEvent	изменение модели таблицы
TreeExpansionEvent	свертывание и развертывание элементов дерева
TreeModelEvent	изменение модели дерева
TreeSelectionEvent	изменение выделенного элемента дерева

Модели данных приложения

Представляют данные, выражающиеся обычно количественно (например, значение в ячейке таблицы). Для таких моделей рекомендуется обязательное разделение логического и визуального представления.

К ним относятся:

Модель	Компоненты
ComboBoxModel	JComboBox
ListModel	JList
TableModel	JTable
TreeModel	JTree
Document	JeditorPane,JTextPane,JTextArea, JTextField,JPasswordField

Модели состояния GUI

Определяют состояние графического компонента (кнопка нажата/не нажата). При их использовании не обязательно осуществлять раздельное управление

моделью, так как работа с моделями состояния GUI осуществляется неявно с помощью методов компонентов.

К этим моделям относятся:

Модель	Компоненты
`ButtonModel`	`JButton,JMenu,JMenuItem`
`SingleSelectionModel`	`JTabbedPane`
`ListSelectionModel`	`JList`
`TableColumnModel`	`JTable`
`TreeSelectionModel`	`Jtree`

Смешанные модели

К ним относятся модели, которые частично представляют собой данные, а частично - состояние компонента. К ним относятся:

Модель	Компоненты
`ButtonModel`	`JToggleButton,JCheckBox,JRadioButton, JCheckBoxMenuItem,JRadioButtonMenuItem`
`BoundedRangeModel`	`JProgressBar,JScrollBar,JSlider`

Модели по умолчанию

Модель, связанная с компонентом, может быть определена как явным образом с помощью метода `setModel()`, так и автоматически. При создании компонента с ним связывается соответствующая модель *по умолчанию*. Например, при создании `JSlider` с ним связывается модель `DefaultBoundedRangeModel`.

Для более сложных объектов, таких как `JTable` или `JTree`, в пакет Swing кроме моделей по умолчанию `DefaultTableModel` и `DefaultTreeModel` входят *абстрактные* модели, которые реализуют часть функций, связанных с извещением об изменении данных. Абстрактные модели имеют имена, где слово `Default` заменено на `Abstract` (например `AbstractTableModel`).

Извещение об изменениях модели

Для сообщения об изменении состояния модели используется механизм событий. Извещение об изменении состояния модели может быть осуществлено двумя способами:

- Посылка легковесного извещения о том, что состояние изменилось, и требование дополнительного запроса от слушателя о том, что именно изменилось. Преимуществом данного подхода является возможность

использования только одного экземпляра события для всех последовательных извещений от конкретной модели. Это удобно, когда события происходят с достаточно высокой частотой.

- Посылка полноопределенного извещения, где детально описывается как изменилось состояние модели. При этом для каждого события должен быть создан отдельный экземпляр события. Такой способ применяется в случаях, когда обычное извещение не может обеспечить слушателя информацией о том, что именно изменилось в модели.

Легковесное извещение

Модель	Слушатель	Событие
`BoundedRangeModel`	`ChangeListener`	`ChangeEvent`
`ButtonModel`	`ChangeListener`	`ChangeEvent`
`SingleSelectionModel`	`ChangeListener`	`ChangeEvent`

Событие `ChangeEvent`, содержащее источник события, посылается классу, воплощающему интерфейс `ChangeListener`. Этот интерфейс определяет единственный метод `stateChanged(ChangeEvent e)`. Для регистрации слушателя используется метод `addChangeListener()`. Получить источник события для последующих запросов можно с помощью метода getSource() класса ChangeEvent.

Полноопределенное извещение

Модель	Слушатель	Событие
`ListModel`	`ListDataListener`	`ListDataEvent`
`ListSelectionModel`	`ListSelectionListener`	`ListSelectionEvent`
`ComboBoxModel`	`ListDataListener`	`ListDataEvent`
`TreeModel`	`TreeModelListener`	`TreeModelEvent`
`TreeSelectionModel`	`TreeSelectionListener`	`TreeSelectionEvent`
`TableModel`	`TableModelListener`	`TableModelEvent`
`TableColumnModel`	`TableColumnModelListener`	`TableColumnModelEvent`
`Document`	`DocumentListener`	`DocumentEvent`
`Document`	`UndoableEditListener`	`UndoableEditEvent`

В этом случае используются специфические события и слушатели для каждой модели. Их обработка аналогична обработке событий при легковесном извещении, за исключением того, что запросы об изменении состояния делаются не у источника события, а у самого события, которое в данном случае содержит полную информацию о всех изменениях модели.

Деревья JTree

Сложные деревья строятся реализацией интерфейса `TreeModel`. В простейшем случае можно поступить следующим образом:

```
public class JTreePanel extends JPanel {
  JTreePanel() {
  setLayout(new BorderLayout());
  //
  DefaultMutableTreeNode root =
    new DefaultMutableTreeNode("Contacts");
  DefaultMutableTreeNode level1 =
    new DefaultMutableTreeNode("Business");
  root.add(level1);
  DefaultMutableTreeNode level2 =
    new DefaultMutableTreeNode("JavaSoft");
  level1.add(level2);

  level2.add(new DefaultMutableTreeNode("James Gosling"));
  level2.add(new DefaultMutableTreeNode("Frank Yellin"));
  level2.add(new DefaultMutableTreeNode("Tim Lindholm"));
  level2 = new DefaultMutableTreeNode("Disney");
  level1.add(level2);
  level2.add(new DefaultMutableTreeNode("Goofy"));
  level2.add(new DefaultMutableTreeNode("Mickey Mouse"));
  level2.add(new DefaultMutableTreeNode("Donald Duck"));

  JTree tree = new JTree(root);
  JScrollPane pane = new JScrollPane();
  pane.getViewport().add(tree);
  add(pane, BorderLayout.CENTER);
  }
}
```

Работа с таблицами (JTable)

Пример создания таблицы с созданием собственной модели таблицы (`MyTableModel`), реализующей интерфейс `TableModel` в виде абстрактного класса — шаблона модели:

```
public class SimpleTableDemo extends JPanel {
  public SimpleTableDemo() {
    JTable table = new JTable(new MyTableModel());
    JScrollPane scrollPane = new JScrollPane(table);
    scrollPane.setPreferredSize(new Dimension(400, 100));
```

```
    setLayout(new GridLayout(1, 0)); add(scrollPane);
  }

class MyTableModel extends AbstractTableModel {
  // данные
  final String[] columnNames = {"First Name", "Last Name",
    "Sport", "Est. Years Experience"};
  final String[][] data = {
    {"Mary", "Campione", "Snowboarding", "5"},
    {"Alison", "Huml", "Rowing", "3"},
    {"Kathy", "Walrath", "Chasing toddlers", "2"},
    {"Mark", "Andrews", "Speed reading", "20"},
    {"Angela", "Lih", "Teaching high school", "4"}
  };
  // реализация TableModel — размеры таблицы
  public int getColumnCount() {return columnNames.length;}
  public int getRowCount() {return data.length;}
  public String getColumnName(int col)
         {return columnNames[col];}
  // реализация TableModel — данные в ячейке
  public Object getValueAt(int row, int col)
         {return data[row][col];}
}
  . . .
}
```

Обобщенное программирование

Обобщенное программирование (generic programming) – описание данных и алгоритмов в программе, которое можно применить к различным типам данных, не меняя при этом само описание.

Для такого описания используются специальные синтаксические конструкции, называемые *шаблонами (дженериками)*.

Шаблоны (generics)

Шаблоны – описание класса, метода, атрибута без использования конкретного типа данных.

Без шаблонов	С шаблонами

Объявление:
`List l = new LinkedList();`
`l.add(new Integer(0));`
`//потенциальна ошибка (*)`
`l.add(new Double(1.1));`
 	
Использование:
`for(…) {`
`  Integer x = (Integer)`
`  l.iterator().next();`
`}`
	Объявление:
	`List<Integer> l =`
	`  new LinkedList<Integer>();`
	`l.add(new Integer(0));`
	`//ошибка компиляции`
	`l.add(new Double(1.1);`
	
	Использование:
	`for (…) {`
	`  Integer x`
	`=l.iterator().next();`
	`}`

Шаблоны повышают наглядность кода и снижают количество явных преобразований типа и возможных ошибок от неявных преобразований.

В примере с шаблонами отсутствует явное приведение к типу `Integer`. Это исключает появление ошибки `ClassCastException` в момент работы программы (* - при ошибочном добавлении в `List` элемента `Double`), а также упрощает визуальное восприятие *доступа к элементам* и делает проще замену типа данных `Integer` на, например, `Double`.

Синтаксические конструкции с использованием шаблонов запрещены в перечислениях, исключительных ситуациях и анонимных встроенных классах.

При компиляции программы происходит уничтожение информации о шаблонах (*type erasure*) и приведение всех *обобщенных* и *параметризованных* типов, *формальных параметров типа* к использованию только базового типа.

Пример:

```
System.out.println("ArrayList<String> это "
     +new ArrayList<String>().getClass());
System.out.println("ArrayList<Double> это "
     +new ArrayList<Double>().getClass());
```

Выдаст:

```
ArrayList<String> это class java.util.ArrayList
ArrayList<Double> это class java.util.ArrayList
```

Основное применение шаблонов — *коллекции*.

Описание типов с шаблонами (generic type)

Обобщенный тип – это описание класса с использованием формальных параметров типа.

Параметризованный тип — реализация обобщенного типа с использованием конкретного типа данных в качестве аргумента.

описание класса без шаблонов	описание с использованием шаблонов

`class Game {`
`  int result;`
`  int getResult();`
`  void setResult(int result);`
`}`
	`class Game<T> {`
	`  T result;`
	`  T getResult();`
	`  void setResult(T result);`
	`}`

`Game<T>` - обобщенный тип

`T` – формальный параметр типа

`Game<String> g = new Game<String>()` - параметризованный тип для представления результатов игры, например, в футбол ("2:0"), а `Game<Integer> g = new Game<Integer>()` в тетрис.

Описание методов с шаблонами (generic methods)

Метод с использованием формального описания типа называется *шаблонным методом*.

Признаком того, что метод является шаблонным служит указание типа данных с которым работат метод. В нашем примере это символ **<T>**

Объявление:

```
public static <T> Set<T> emptySet() {
        return new HashSet<T>();
}
```

Вызов:

```
// конкретный тип для подстановки выбирает компилятор по
// аргументам вызова метода или оператору присвоения
Set<String> = Collections.emptySet();
// указан явно
Set<Integer> = Collections.<Integer>emptySet(); //
```

Формальные параметры типа (type parameter)

Формальный параметр типа — это параметр, вместо которого при создании по шаблону параметризованного типа необходимо подставить конкретный тип данных.

```
interface Comparable<E> {
    int compareTo(E other);
}
```

в приведенном примере `E` является формальным параметром типа. Формальных параметров может быть несколько — `KeyValue<KEY,value>`.

Формальные параметры могут быть *ограниченными*.

Ограниченный формальный параметр позволят задать возможные границы подстановки конкретных типов данных для получения параметризованного типа

Ограничение могут быть с верхней и с нижней стороны дерева наследования:

- `<T extends Number>` - ограничивает T сверху — в качестве формального параметра могут быть использованы только потомки класса `Number;`
- `<T super Integer>` - ограничивает T снизу — могут быть использованы только предки класса `Integer.`

В ограничении могут быть использованы символы '&' и '|' для объединения условий соответствующими логическими операциями.

Wildcard

> *Wildcard (дословно джокер) или групповая подстановка* – синтаксические конструкции с использованием символа '?' (означает *любой тип*), используемые для замены в шаблонах конкретного класса множеством возможных классов.

Групповая подстановка обычно используется для создания шаблонных классов:

```
/*
 * Допустимые параметры:
 * Square s (Square extends Shape) — можно
 * String str — нельзя
 */
static void drawAll(Collection<? extends Shape> c) {
  for(Shape s : c) {
    s.draw();
  }
}
```

Коллекции

> *Коллекции* – это классы для сохранения, получения, манипулирования данными над множеством объектов.

Иерархия коллекций представлена на рис. 23.

Iterator

> *Iterator* – интерфейс, позволяющий выполнять обход элементов коллекции с возможностью их удаления. Является переработкой и развитием интерфейса `Enumeration`.

`boolean hasNext();`	Возвращает `true` если в коллекции есть еще элементы.
`E next();`	Следующий элемент.
`void remove();`	Удалить элемент.

Использование `Iterator` вместо `Enumeration` является предпочтительным из-за более короткого описания методов и появившейся операции remove. Класс `ListIterator` осуществляет двунаправленное перемещение по коллекции.

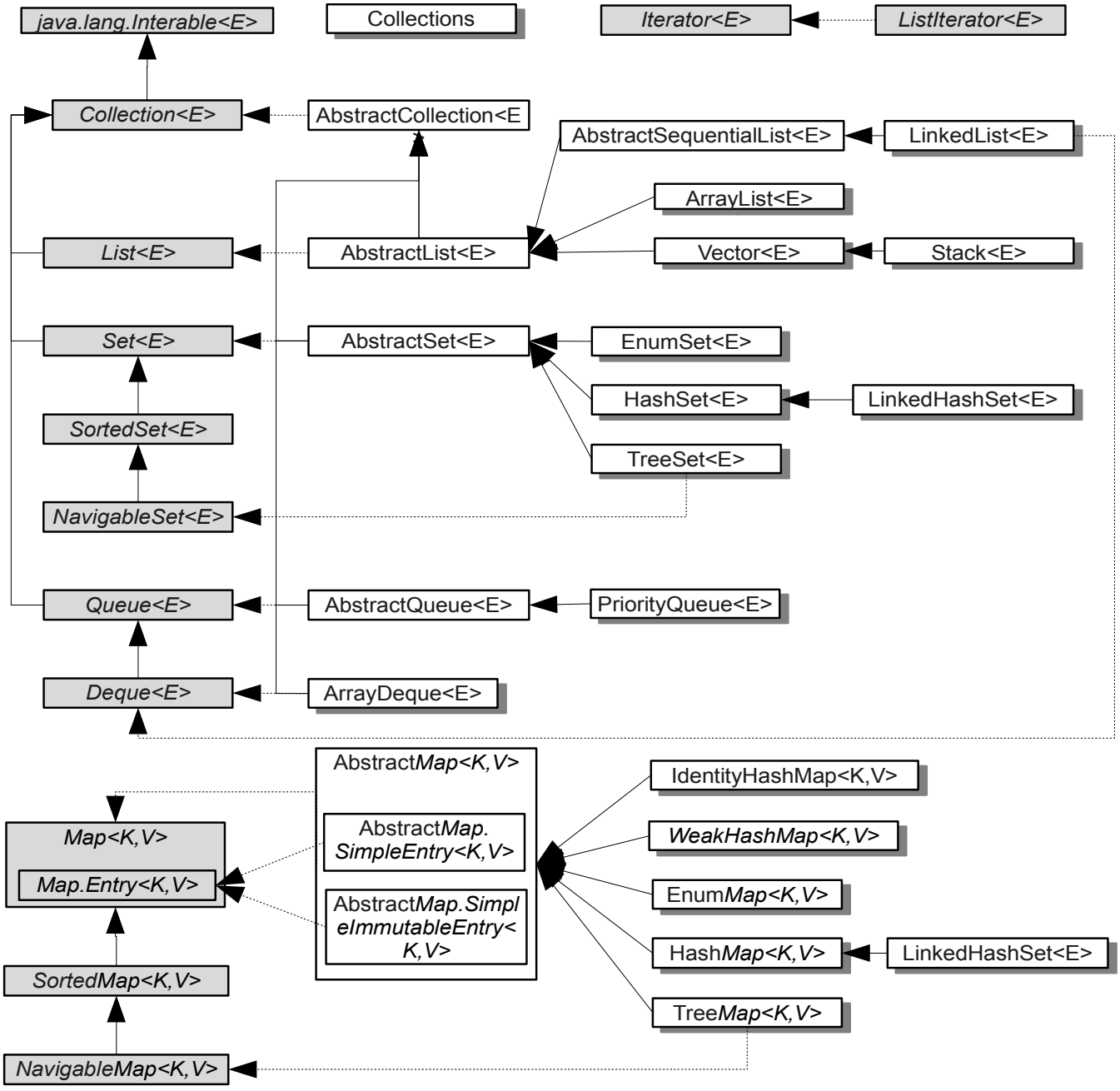

Рисунок 23. Иерархия коллекций.

Collection

Collection – интерфейс, описывающий набор каких-либо элементов. Способ организации элементов определяется реализующими Collection классами и интерфейсами.

`int size()`	Возвращает размер коллекции.
`boolean isEmpty()`	Возвращает `true`, если в коллекции нет элементов.
`boolean contains(Object o)`	Возвращает `true`, если коллекция содержит элемент.
`Iterator<E> iterator();`	Возвращает `Iterator`.
`Object[] toArray();`	Преобразует коллекцию в массив объектов.
`<T> T[] toArray(T[] a);`	Преобразует в коллекцию в массив типа `T`, размещая в массиве `a`.
`boolean add(E e);`	Добавляет элемент в коллекцию. Возвращает `true`, если элемент добавлен; `false` — если коллекция запрещает дубликаты элементов и элемент не уникален.
`boolean remove(Object o);`	Удаляет *один* элемент (даже если таких несколько). Возвращает `true`, если элемент удален.
`boolean containsAll(Collection<?> c);`	Возвращает `true`, если коллекция содержит все элементы заданной коллекции.
`boolean addAll (Collection<? extends E> c);`	Добавляет элементы в коллекцию. Возвращает `true`, если коллекция изменена в результате операции.
`boolean removeAll (Collection<?> c);`	Удаляет все элементы, заданные в коллекции `c`.
`boolean retainAll (Collection<?> c);`	Удаляет из коллекции все элементы, кроме тех, которые заданы в коллекции `c`.
`void clear();`	Удаляет из коллекции все элементы.
`boolean equals(Object o);`	Проверяет коллекцию на равенство другой коллекции
`int hashCode();`	Возвращает хеш-код коллекции.

В потомках интерфейса `Collection` допускается реализация не всех методов. В таком случае необходимо генерировать исключительную ситуацию `UnsupportedOperationException`.

Любая коллекция должна возвращать итератор, и соответственно, позволять обходить свои элементы.

Интерфейсы коллекций

`Set`	Множество. Элементы уникальны и, возможно, отсортированы.
`List`	Сортированная последовательность элементов, с возможностью дублирования и позиционного доступа.
`Queue`	Очередь, предназначенная для размещения элемента перед его обработкой. Расширяет коллекцию методами для вставки, выборки и просмотра элементов.
`Deque`	Двунаправленная очередь, позволяющая вставку и удаление в два конца очереди.
`Map`	Карта это соответствие ключ — значение. Каждому ключу соответствует одно значение.
`SortedSet`	Множество, элементы которого автоматически сортируются либо в их натуральном порядке (интерфейс `Comparable`), либо с использованием `Comparator`.
`SortedMap`	Автоматически сортированная карта (см. `SortedSet`)
`Navigable Set`	`SortedSet`, расширенный методами кратчайшего доступа к искомому элементу. В `NavigableSet` элементы могут быть доступны в порядке увеличения и уменьшения
`Navigable Map`	`SortedMap`, расширенный методами кратчайшего доступа к искомому элементу

В пакете `java.util.concurrent` доступны дополнительные интерфейсы `BlockingQueue`, `BlockingDeque` (ограниченное число элементов и ожидание освобождения места), `ConcurrentMap`, `ConcurrentNavigableMap` (атомарные операции вставки, удаления, замены).

Для уменьшения трудоемкости реализации существуют общие интерфейсы реализации коллекций - `AbstractCollection`, `AbstractMap` и др. представляющие тривиальную реализацию основных методов коллекций.

Коллекции общего назначения

`HashSet`	Реализация множества с использованием хеш-таблиц, обладающая самым высоким быстродействием
`TreeSet`	Реализация `NavigableSet` интрефейса с использованием раскрашеных деревьев

`LinkedHashSet`	Реализация множества в виде хеш таблицы и дву-связанного списка с заданным порядком элементов
`ArrayList`	Массив переменного размера. Является потоко-небезопасным `Vector`. Наиболее высокое быстродействие
`ArrayDeque`	Эффективная реализация интерфейса `Deque` переменного размера
`LinkedList`	Двусвязная реализация интерфейса `List`. Быстрее `ArrayList`, если элементы часто добавляются и удаляются. В дополнение реализует интерфейс `Deque`
`PriorityQueue`	Реализация очереди с приоритетами
`HashMap`	Реализация интерфейса `Map` с использованием хеш таблиц
`TreeMap`	Реализация `NavigableMap` интрефейса с использованием раскрашеных деревьев
`LinkedHashMap`	Реализация карты в виде хеш таблицы и дву-связанного списка с заданным порядком элементов

В дополнение к общим реализациям, существуют прародители коллекций - классы `Vector` и `Hashtable`, реализация которых была обновлена с использованием шаблонов.

Специальные коллекции

`WeakHashMap`	Карта, сохраняющие слабые ссылки на ключи. Позволяет сборщику мусора уничтожить пару ключ-значение, когда на на ключ более нет внешних ссылок
`IdentityHashMap`	Реализация интерфейса `Map` с использованием хеш таблицы и сравнением объекта на равенстов по ссылке (`key1==key2`), вместо сравнения по значению (`key1.equials(key2)`).
`CopyOnWriteArrayList`	Реализация List, в которой операции - мутаторы (`add, set, remove`) реализуются путем создания новой копии `List`. В результате нет необходимости в синхронизации.
`CopyOnWriteArraySet`	Реализация Set с созданием новой копии по операции изменения (см. `CopyOnWriteArrayList`)
`EnumSet`	Высокопроизводительная реализация множества с использованием битового вектора. Все элементы должны быть элементами одного `Enum`
`EnumMap`	Высокопроизводительная реализация карты с

	использованием битового вектора. Все ключи должны быть элементами одного `Enum`

Пакета `java.util.concurrent` дополняет реализации коллекций классами `ConcurrentLinkedQueue`, `LinkedBlockingQueue`, `ArrayBlockingQueue`, `PriorityBlockingQueue`, `DelayQueue`, `SynchronousQueue`, `LinkedBlockingDeque`, `ConcurrentHashMap`, `ConcurrentSkipListSet`, `ConcurrentSkipListMap`, которые подготовлены для использования в многопоточных программах и реализуют различную дополнительную функциональность.

Сортировка элементов коллекции

Сортировка элементов коллекции в интерфейсе `SortedMap` и аналогичных производится при помощи естественного порядка сортировки, определяемого в элементе коллекции, либо при помощи интерфейса `Comparator`.

> *Естественный порядок сортировки (natural sort order)* — естественный и реализованный по умолчанию (реализацией метода `compareTo` интерфейса `java.lang.Comparable`) способ сравнения двух экземпляров одного класса.

`int compareTo(E other)` — сравнивает `this` объект с `other` и возвращает отрицательное значение если `this<other`, 0 — если они равны и положительное значение если `this>other`. Для класса `Byte` данный метод реализуется следующим образом:

```
public int compareTo(Byte anotherByte) {
     return this.value - anotherByte.value;
}
```

`java.util.Comparator` — содержит два метода:

- `int compare(T o1, T o2)` — сравнение, аналогичное compareTo
- `boolean equals(Object obj)` — true если `obj` это `Comparator` и у него такой же принцип сравнения.

Collections

> Collections — класс, состоящий из статических методов, осуществляющих различные служебные операции над коллекциями.

`sort(List)`	Сортировать список, используя merge sort алгоритм, с гарантированной скоростью O (n*log n).

`binarySearch(List, Object)`	Бинарный поиск элементов в списке.
`reverse(List)`	Изменить порядок элементов в списке на противоположный.
`shuffle(List)`	Случайно перемешать элементы.
`fill(List, Object)`	Заменить каждый элемент заданным.
`copy(List dest, List src)`	Скопировать список `src` в `dst`.
`min(Collection)`	Вернуть минимальный элемент коллекции.
`max(Collection)`	Вернуть максимальный элемент коллекции.
`rotate(List list, int distance)`	Циклически повернуть список на указанное число элементов.
`replaceAll(List list, Object oldVal, Object newVal)`	Заменить все объекты на указанные.
`indexOfSubList(List source, List target)`	Вернуть индекс первого подсписка `source`, который эквивалентен `target`.
`lastIndexOfSubList(List source, List target)`	Вернуть индекс последнего подсписка `source`, который эквивалентен `target`.
`swap(List, int, int)`	Заменить элементы в указанных позициях списка.
`unmodifiableCollection (Collection)`	Создает неизменяемую копию коллекции. Существуют отдельные методы для `Set`, `List`, `Map`, и т.д.
`synchronizedCollection (Collection)`	Создает потоко-безопасную копию коллекции. Существуют отдельные методы для `Set`, `List`, `Map`, и т.д.
`checkedCollection (Collection<E> c, Class<E> type)`	Создает типо-безопасную копию коллекции, предотвращая появление неразрешенных типов в коллекции. Существуют отдельные методы для `Set`, `List`, `Map`, и т.д.
`<T> Set<T> singleton(T o);`	Создает неизменяемый `Set`, содержащую только заданный объект. Существуют методы для `List` и `Map`.
`<T> List<T> nCopies(int n, T o)`	Создает неизменяемый `List`, содержащий n копий заданного объекта.
`frequency(Collection, Object)`	Подсчитать количество элементов в коллекции.

`reverseOrder()`	Вернуть `Comparator`, которые предполагает обратный порядок сортировки элементов.
`list(Enumeration<T> e)`	Вернуть `Enumeration` в виде `ArrayList`.
`disjoint(Collection, Collection)`	Определить, что коллекции не содержат общих элементов.
`addAll(Collection<? super T>, T[])`	Добавить все элементы из массива в коллекцию.
`newSetFromMap(Map)`	Создать `Set` из `Map`.
`asLifoQueue(Deque)`	Создать `Last` in first out `Queut` представление из `Deque`.

В 2009 году Университет стал победителем многоэтапного конкурса, в результате которого определены 12 ведущих университетов России, которым присвоена категория «Национальный исследовательский университет». Министерством образования и науки Российской Федерации была утверждена Программа развития государственного образовательного учреждения высшего профессионального образования «Санкт-Петербургский государственный университет информационных технологий, механики и оптики» на 2009–2018 годы.

КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

Кафедра Вычислительной техники СПбГУ ИТМО создана в 1937 году и является одной из старейших и авторитетнейших научно-педагогических школ России.

Первоначально кафедра называлась кафедрой математических и счетно-решающих приборов и устройств и занималась разработкой электромеханических вычислительных устройств и приборов управления. Свое нынешнее название кафедра получила в 1963 году.

Кафедра вычислительной техники является одной из крупнейших в университете, на которой работают высококвалифицированные специалисты, в том числе 8 профессоров и 15 доцентов, обучающие около 500 студентов и 30 аспирантов.

Гаврилов Антон Валерьевич
Клименков Сергей Викторович
Цопа Евгений Алексеевич

Программирование на Java

Конспект лекций

В авторской редакции
Редакционно-издательский отдел Санкт-Петербургского государственного университета информационных технологий, механики и оптики
Зав. РИО Н.Ф. Гусарова
Лицензия ИД № 00408 от 05.11.99
Подписано к печати
Заказ №
Тираж
Отпечатано на ризографе

Редакционно-издательский отдел
Санкт-Петербургского государственного
университета информационных технологий,
механики и оптики
197101, Санкт-Петербург, Кронверкский пр., 49