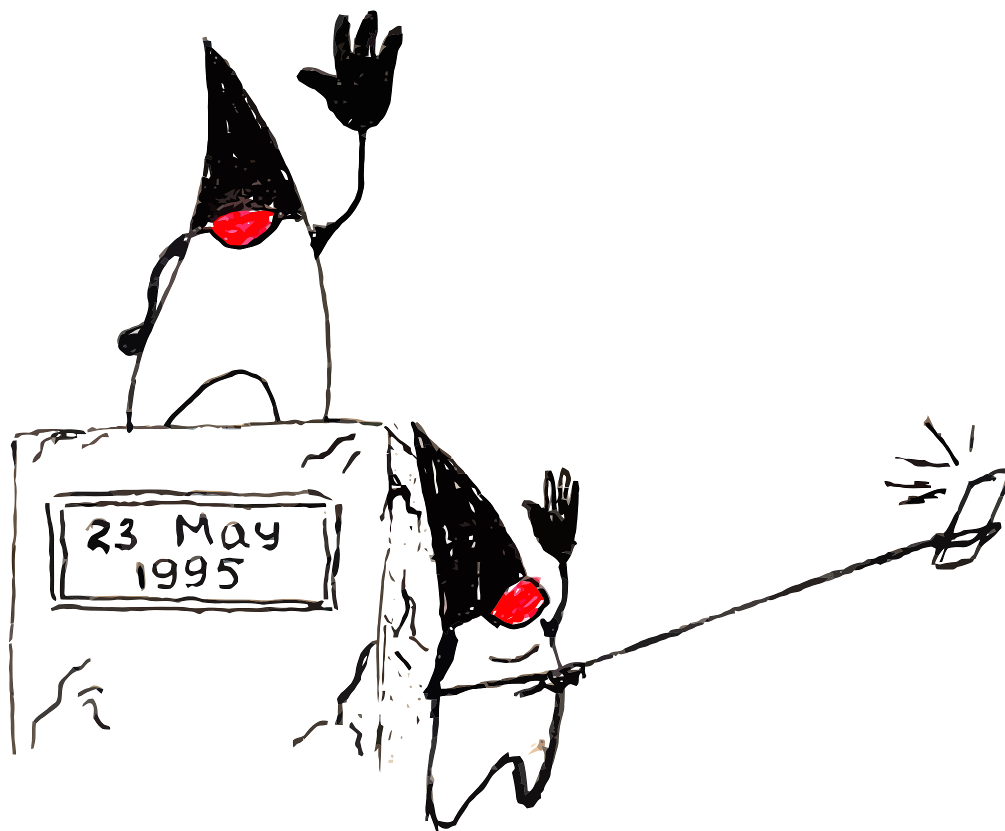
УНИВЕРСИТЕТ ИТМО

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ JAVA



Санкт-Петербург

2015

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

УНИВЕРСИТЕТ ИТМО

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

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ JAVA

Конспект лекций

Санкт-Петербург

2015

Гаврилов А.В., Клименков С.В., Харитонова А.Е., Цопа Е.А.
Программирование на языке Java. Конспект лекций – СПб: Университет ИТМО, 2015. – 126 с.

Данное пособие представляет собой краткий справочник по языку Java и может использоваться как конспект лекционного курса «Программирование 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 по направлениям 230100.62.01 «Вычислительные машины, комплексы, системы и сети», 231000.62.01 «Разработка программно-информационных систем», бакалавров по направлениям 230100.62 «Информатика и вычислительная техника», 231000.62 - «Программная инженерия».

Рекомендовано к печати Ученым советом факультета КТиУ, протокол №10 от 15.12.2015.

Университет ИТМО – ведущий вуз России в области информационных и фотонных технологий, один из немногих российских вузов, получивших в 2009 году статус национального исследовательского университета. С 2013 года Университет ИТМО – участник программы повышения конкурентоспособности российских университетов среди ведущих мировых научно-образовательных центров, известной как проект «5 в 100». Цель Университета ИТМО – становление исследовательского университета мирового уровня, предпринимательского по типу, ориентированного на интернационализацию всех направлений деятельности.

© Университет ИТМО, 2015

©Гаврилов А.В, Клименков С.В., Харитонова А.Е, Цопа Е.А., 2015

Оглавление

Введение. Основные концепции ООП...5
Язык программирования Java..5
Объекты и классы..5
Конструкторы..6
Сообщения..7
Инкапсуляция...7
Наследование..8
Полиморфизм...9
Интерфейсы..9
Вложенные, локальные и анонимные классы...13
Принципы проектирования классов...14
Инструментальные средства JDK 1.8..16
javac..17
java..18
javadoc..18
appletviewer..20
Интегрированные среды разработки...21
Синтаксис языка Java..22
Приложение Hello, World!..22
Апплет Hello, World!..23
Класс Applet...23
Безопасность при работе с апплетами..24
Лексический разбор исходного кода...24
Идентификаторы и ключевые слова..25
Типы данных и литералы...25
Переменные...27
Области видимости...28
Операторы..29
Управляющие конструкции..30
Модификаторы...30
Аннотации..31
Перечисляемые типы...32
Использование объектов...32
Метод finalize()..32
Стандартная библиотека классов..33
Пакеты, входящие в JDK 1.8...33
Пакет java.lang...36
Класс Object...37
Класс Class...37
Класс System..38
Класс Math..38
Классы-оболочки..38
Автоупаковка и автораспаковка...38
Класс String..39

Классы StringBuffer и StringBuilder......39
Использование String и StringBuffer......40
Класс java.util.StringTokenizer......40
Класс Exception......41
Класс RuntimeException......42
Класс Error......43
Множественная обработка исключений......43
Проброс исключений более узких типов......44
Выражение try-with-resources......45
Пакет java.util......48
Использование легковесных процессов......50
Реализация потока......50
Состояние потока......51
Распределение приоритета между потоками......51
Класс java.lang.ThreadGroup......51
Методы класса java.lang.Thread......52
Взаимодействие и синхронизация потоков......52
Модификатор volatile......54
Обобщенное программирование......55
Шаблоны......55
Описание типов с шаблонами......56
Описание методов с шаблонами......56
Формальные параметры типа......57
Шаблоны с групповой подстановкой......57
Коллекции......59
Интерфейс Iterator......60
Интерфейс Collection......60
Интерфейсы коллекций......61
Коллекции общего назначения......62
Специальные коллекции......62
Сортировка элементов коллекции......63
Класс Collections......64
Лямбда-выражения......66
Пакет java.util.concurrent......71
Исполнители (Executors)......71
Очереди (Queues)......73
Потокобезопасные коллекции (Concurrent Collections)......74
Синхронизаторы (Synchronizers)......74
Блокировки (Locks)......75
Атомарные объекты (Atomic objects)......77
Работа с потоками ввода-вывода......78
Иерархия потоков в Java......78
Класс InputStream......78
Класс OutputSrteam......79
Класс Reader......80
Класс Writer......80

Специализированные потоки....................81
Преобразующие потоки....................82
Стандартные потоки ввода-вывода....................83
Сериализация объектов....................84
Интерфейс java.io.Serializable....................84
Класс java.io.ObjectOutputStream....................84
Класс java.io.ObjectInputStream....................85
Пример сериализации и восстановления объекта....................85
Интерфейс java.io.Externalizable....................85
Контроль версий сериализуемого класса....................86
Основы сетевого взаимодействия....................87
Работа с адресами....................87
Передача данных по протоколу TCP....................88
Передача данных по протоколу UDP....................90
Работа с URL-соединениями....................92
Расширенный ввод-вывод....................93
Работа с буферами....................93
Кодировки символов....................95
Каналы....................96
Файловые каналы....................96
Сетевые каналы....................96
RMI – вызов удаленных методов....................98
Структура RMI....................98
Определения....................98
Определение удаленных интерфейсов....................99
Создание сервера....................99
Создание клиентов....................100
Запуск каталога, сервера и клиентов....................100
Интернационализация и локализация....................100
Интернационализация....................100
Локализация....................101
Класс Locale....................102
Класс ResourceBundle....................103
Класс ListResourceBundle....................103
Класс PropertyResourceBundle....................104
Иерархия классов java.text....................104
Класс NumberFormat....................105
Класс DecimalFormat....................106
Класс DecimalFormatSymbols....................107
Класс DateFormat....................107
Класс SimpleDateFormat....................108
Класс DateFormatSymbols....................109
Класс MessageFormat....................109
Класс ChoiceFormat....................111
Класс Collator....................111
Класс RuleBasedCollator....................112

Класс CollationKey....113
Рефлексия....114
Класс Class....114
Интерфейс Member....115
Класс Modifier....115
Класс Field....116
Класс Method....116
Класс Constructor....117
Класс Array....117

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,	лает, кусает, виляет хвостом, гры-

	цвет	зет 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 инструкции **new.** При этом происходит выделение памяти для хранения переменных объекта.

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(); // создание объекта`
	`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) {`
	`Point a = new Point();`
	`a.isVisible = true;`
	
	`Point b = new Point(20,50);`

```
    x = x0;
    y = y0;
  }
}
```

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;
  }
 	
  void show() {
    isVisible = true;
  }
  void move(x1, y1) {
    x = x1;
    y = y1;
  }
}
	Point a = new Point(20,30);
	a.isVisible = true;
	
	a.hide();
	a.move(60,80);
	a.show();
	
	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

```
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;
  }
}
```

Доступ к переменным класса

```
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

```
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;
  }
```

Наследование

```
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;
```

<table>
<tr><td><pre>}</pre></td><td><pre> }
}</pre></td></tr>
</table>

<table>
<tr><td><pre>Point a = new Point(30,40);
int ax, ay;
ax = a.getX();
ay = a.getY();</pre></td><td><pre>Pixel b = new Pixel(20,50);
b.setColor(2);
int bx, by, bc;
bx = b.getX();
by = b.getY();
bc = b.getColor();</pre></td></tr>
</table>

Полиморфизм

Полиморфизм — возможность единообразно обрабатывать объекты разных типов. Раличают *полиморфизм включения* (или полиморфизм подтипов), который позволяет обращаться с помощью единого интерфейса к классу и к любым его потомкам, и *параметрический полиморфизм*, который определяет обобщенный интерфейс с одинаковой реализацией.

<table>
<tr><th>Объявления классов</th><th>Использование полиморфизма</th></tr>
<tr><td><pre>public class Shape {
  ...
 abstract void draw();
}

public class Rectangle
  extends Shape {
  ...
  void draw() {
    ...
  }
}

public class Circle
  extends Shape {
  ...
  void draw() {
    ...
  }
}</pre></td><td><pre>public class FiguresSet {
  ...

  public static void main
    (String args[]) {
    ...
    Shape rect;
    Shape circ;
    ...
    rect = new Rectangle();
    circ = new Circle();
    ...
    rect.draw();
    circ.draw();
  }
}</pre></td></tr>
</table>

Интерфейсы

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

Объявление интерфейса	*Воплощение интерфейса*

`interface Displayable {`
`  void hide();`
`  void show();`
`}`
	`public class Pixel`
	`  extends Point`
	`  implements Displayable {`
	
	`  ...`
	
	`  void hide() {`
	`    ...`
	`  }`
	`  void show() {`
	`    ...`
	`  }`
	`}`

Класс может воплощать любое количество интерфейсов.

В Java 7 и более ранних версиях интерфейсы могли содержать *только* заголовки методов и константы. Это приводило к проблемам в случае модификации интерфейса – при добавлении к нему нового метода все его реализации "ломались" до тех пор, пока в них не будет добавлена реализация добавленного метода:

```
/*
 * До модификации
 */
public interface Orderable {
  void orderByID();
}

// Класс скомпилируется
public class MyCollection implements Orderable {
  ...
  public void orderByID() {
    ...
  }
  ...
}

/*
 * После модификации
 */
public interface Orderable {
  void orderByID();
  void orderByName(); // Новый метод
}

// Класс не скомпилируется –
// не реализован метод orderByName
public class MyCollection implements Orderable {
  ...
```

```
  public void orderByName() {
    ...
  }
  ...
}
```

Эта проблема была решена в Java 8. Теперь, помимо заголовков методов и констант, интерфейсы могут содержать и конкретные реализации методов. Такие методы называются *методами по умолчанию*; при их объявлении должно быть использовано ключевое слово `default`:

```
public interface Orderable {
  void orderByID();
  default void orderByName() {
    ... // Какая-то реализация
  }
}

// Класс скомпилируется –
// метод orderByName уже реализован в интерфейсе
public class MyCollection implements Orderable {
  ...
  public void orderByName() {
    ...
  }
  ...
}
```

Так как класс может реализовывать множество интерфейсов, может возникнуть ситуация, когда один и тот же метод объявлен или реализован сразу в нескольких из них. Если метод объявлен более, чем в одном из интерфейсов, то программист *обязан* переопределить его в классе, независимо от того, реализован этот метод в интерфейсах, или нет.

<table>
<tr><th>Код</th><th>Комментарий</th></tr>
<tr><td><pre>interface Human {
  default void getName();
}

public class Businessman
  implements Human, Company {
  // Метод getName не реализован
  ...
}</pre></td><td>Код не скомпилируется – требуется реализовать метод <code>getName()</code> в классе.</td></tr>
<tr><td><pre>interface Human {
  default void getName() {
    ... // Реализация
  }
}

public class Businessman</pre></td><td>Код скомпилируется – метод уже реализован в интерфейсе; реализация в классе не нужна.</td></tr>
</table>

<table>
<tr><td><pre> implements Human, Company {
  // Метод getName не реализован
  ...
}</pre></td><td></td></tr>
<tr><td><pre>interface Human {
  default void getName() {
    ... // Реализация
  }
}

interface Company {
  default void getName() {
    ... // Реализация
  }
}

public class Businessman
  implements Human, Company {
  // Метод getName не реализован
  ...
}</pre></td><td>Код не скомпилируется – “конфликт” между интерфейсами. Нужно переопределить метод в классе.</td></tr>
<tr><td><pre>interface Human {
  default void getName() {
    ... // Реализация
  }
}

interface Company {
  void getName();
}

public class Businessman
  implements Human, Company {
  // Метод getName не реализован
  ...
}</pre></td><td>Код не скомпилируется – нужно переопределить метод <code>getName()</code> в классе, несмотря на то, что он уже реализован в интерфейсе <code>Human</code>.</td></tr>
</table>

Если класс “получает по наследству” реализацию метода одновременно из родительского класса и из интерфейса, то всегда применяется реализация метода из класса:

```
public interface Orderable {
  default void orderByName() {
    ... // Реализация
  }
}

// Не реализует интерфейс Orderable
public class OrderedCollection {
  // Сигнатура совпадает с методом в интерфейсе
  public void orderByName() {
```

```
    ... // Какой-то код
  }
}

// Класс скомпилируется – метод orderByName
// будет унаследован от класса OrderedCollection
public class MyCollection extends OrderedCollection
                          implements Orderable {
  // Метод orderByName не реализован
  ...
}
```

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();
```

Принципы проектирования классов

Для того, чтобы избежать наиболее часто возникающих проблем при проектировании объектно-ориентированного приложения, были разработаны принципы, которым рекомендуется следовать при разработке.

Принцип единственной обязанности (SRP – Single Responsibility Principle)

> Класс должен иметь единственную обязанность. При этом не должно возникать более одной причины для изменения класса.

Принцип открытости-закрытости (OCP - Open-Closed Principle)

> Классы должны быть открыты для расширения, но закрыты для модификации.

Принцип подстановки Барбары Лисков (LSP - Liskov Substitution Principle)

> Наследующий класс должен дополнять, а не замещать поведение базового класса. При этом замена в коде объектов класса-предка на объекты класса-потомка не приведёт к изменениям в работе программы.

Принцип разделения интерфейса (ISP – Interface Segregation Principle)

> Вместо одного универсального интерфейса лучше использовать много специализированных.

Принцип инверсии зависимостей (DIP – Dependency Inversion Principle)

> Зависимости внутри системы строятся на уровне абстракций. Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Абстракции не должны зависеть от деталей, а наоборот, детали должны зависеть от абстракций.

Принцип повышения связности

> *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.

Принцип уменьшения связанности

> *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.

Инструментальные средства JDK 1.8

Основные	
`javac`	компилятор
`java`	интерпретатор
`jdb`	отладчик
`javah`	генератор файлов заголовков и исходных текстов на C
`javap`	дизассемблер классов
`javadoc`	генератор документации
`appletviewer`	программа просмотра апплетов
`jar`	упаковщик классов в исполняемый архив
`extcheck`	детектор конфликтов версий
`jdeps`	анализатор зависимостей
Средства безопасности	
`keytool`	менеджер ключей и сертификатов
`policytool`	программа для управления файлами политик
`jarsigner`	утилита для подписи архивов
Средства интернационализации	
`native2ascii`	конвертер в кодировку Latin-1
Средства поддержки RMI, IDL, RMI-IOOP и веб-сервисов	
`rmic`	RMI-компилятор
`rmid`	RMI-сервер
`rmiregistry`	реестр RMI-объектов
`serialver`	вывод serialVersionUID классов в CLASSPATH
`tnameserv`	служба имён
`idlj`	генератор зависимостей для языка IDL
`orbd`	ORB-сервер
`servertool`	консоль администратора ORB-сервера
`schemagen`	генератор XML-схем
`wsgen`	генератор stubs для веб-сервисов
`wsimport`	генератор ties для веб-сервисов
`xjc`	генератор классов на основе XML-схем
Средства развертывания и запуска приложений	
`javapackager`	упаковщик Java и JavaFX-приложений
`pack200`	программа компрессии архивов jar
`unpack200`	программа для распаковки архивов
`javaws`	программа, осуществляющая загрузку и запуск приложений с удалённых web-серверов
Средства отладки, мониторинга и профилирования	
`jcmd`	утилита команд диагностики
`jconsole`	консоль, предназначенная для мониторинга и управления исполнением приложений

`jmc`	коммерческое средство мониторинга без влияния на производительность
`jvisualvm`	графический профилировщик приложений
`jps`	вывод информации о запущенных процессах
`jstat`	утилита мониторинга статистики
`jstatd`	демон статистики
`jmap`	вывод карты памяти процесса
`jinfo`	вывод информации о конфигурации
`jhat`	браузер дампа кучи
`jstack`	вывод состояния стека
`jsadebugd`	демон для отладки процессов
Средства поддержки скриптов	
`jjs`	запуск скриптов Nashorn
`jrunscript`	**командная строка для скриптов**

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.

Пример установки переменной CLASSPATH в UNIX:

```
CLASSPATH=.:/usr/local/java/swing/classes
export CLASSPATH
```

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.

`-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.

`-jar`

Выполняет приложение, упакованное в архив

`-D`*`имя=значение`*

Присваивает системному свойству с заданным именем указанное значение.

`-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.

`-J`*аргумент*

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>
```

Атрибуты тега <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.

Атрибуты тега <PARAM>

NAME

Имя параметра.

VALUE

Значение параметра.

Интегрированные среды разработки

- NetBeans (Oracle)

  http://www.netbeans.org
- Eclipse (Eclipse Foundation)

  http://www.eclipse.org
- IntelliJIDEA (JetBrains)

  http://www.jetbrains.com/idea/

Синтаксис языка Java

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

Лексический разбор исходного кода

Исходный код программы на языке Java состоит из последовательности символов стандарта Unicode в кодировке UTF-16. Программа также может быть записана в других кодировках, например ASCII. При этом остальные символы Unicode записываются в виде `\uXXXX`, где `XXXX` – шестнадцатеричный код символа UTF-16.

Лексический разбор исходного кода состоит из следующих этапов:

Последовательности вида `\uXXXX` заменяются символами Unicode.

Текст делится на строки. Концом строки считаются символы:

- «перевод строки» (код 10),
- «возврат каретки» (код 13),
- последовательность этих символов (CR LF).

Удаляются комментарии. Комментарии могут быть однострочные и многострочные. Однострочный комментарий — это текст от символов `//` до конца строки. Многострочный комментарий — это текст между символами `/*` и `*/` включительно.

Выделяются лексемы, ограниченные пробельными символами и концами строк. Пробельные символы:

- «пробел» (код 32),
- «табуляция» (код 9),
- «перевод страницы» (код 12).

К лексемам относятся идентификаторы, ключевые слова, литералы, разделители и операторы.

Идентификаторы и ключевые слова

Идентификатор — последовательность букв и цифр, начинающаяся с буквы, не совпадающая с ключевым словом. Идентификаторы используются в качестве имен для элементов программы.

Ключевые слова языка Java перечислены в таблице. В эту же таблицу добавлены литералы (выделены подчеркиванием), которые так же, как и ключевые слова, не могут быть идентификаторами.

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

Типы данных и литералы

Язык Java является языком со строгой статический типизацией. Статическая типизация обозначает, что все переменные и выражения имеют тип, который должен быть известен на этапе компиляции. Строгая типизация подразумевает строгий контроль соответствия типа переменной типу значения, которое ей может быть присвоено. Также тип переменных и выражений определяет набор и семантику операций, которые можно над ними производить.

Типы данных делятся на примитивные и ссылочные. По адресу, связанному с переменной примитивного типа, хранится ее значение. По адресу, связанному с переменной ссылочного типа, хранится ссылка на ее значение.

В языке Java существует 8 примитивных типов данных.

Тип	**Бит**	**Диапазон значений**	**default**
byte	8	[-128 ... 127]	0
short	16	[-32768 ... 32767]	0
int	32	[-2147483648 ... 2147483647]	0
long	64	[-9223372036854775808 ... 9223372036854775807]	0L
char	16	['\u0000' ... '\uffff'] или [0 ... 65535]	'\u0000'
float	32	±[0; 1.4·10^{-45} ... 3.4028235·10^{38}; ∞], NaN	0.0F

double	64	±[0; $4.9\cdot10^{-324}$ ... $1.797693134862315 7\cdot10^{308}$; ∞], NaN	0.0
boolean		false; true	false

К ссылочным типам относятся: классы, интерфейсы, переменные типа и массивы.

Для представления значений переменных используются литералы. Числовые литералы делятся на целые и с плавающей запятой.

Формат целого литерала: (префикс) значение (суффикс). Формат литерала с плавающей запятой: (префикс) целая_часть . дробная_часть основание порядок (суффикс). Варианты обозначения элементов числовых литералов приведены в таблице.

Элемент	**Символ**		**Значение**
префикс	0		восьмеричный литерал (целый)
	0x	0X	шестнадцатеричный литерал
	0b	0B	двоичный литерал (целый)
суффикс	l	L	литерал типа long
	f	F	литерал типа float
	d	D	литерал типа double
основание	e	E	десятичное основание степени
	p	P	двоичное основание степени

Префикс определяет систему счисления литерала — десятичная (без префикса), восьмеричная, шестнадцатеричная или двоичная. Суффикс определяет тип литерала. Литерал без суффикса имеет тип int или double. В литералах могут использоваться заглавные или строчные буквы. Строчный префикс l для типа long использовать не рекомендуется, так как он похож на единицу. Для удобства чтения длинные значения литералов можно делить на группы с помощью символа подчеркивания.

Десятичный литерал с плавающей запятой вида aEb представляет число a·10b, шестнадцатеричный литерал с плавающей запятой вида 0xaPb – число a16·2b. Обязательными элементами литерала с плавающей запятой являются:

- хотя бы одна цифра в целой или дробной части;
- десятичная точка или основание с порядком или суффикс типа для десятичного формата;
- основание с порядком для шестнадцатеричного формата.

Логический литерал имеет тип boolean и может принимать значения false или true.

Символьный литерал представляет собой одиночный символ Unicode, он имеет тип char, его значение задается в одинарных кавычках. Строковый литерал представляет собой последовательность символов Unicode, он имеет тип String и яв-

ляется ссылкой на соответствующий экземпляр класса String. Значение строкового литерала задается в двойных кавычках. Символы Unicode для символьных и стороковых литералов могут задаваться:

- явно;
- в виде \uXXXX, где XXXX – шестнадцатеричный код символа UTF-16;
- в виде \XXX, где XXX – восьмеричный код символа от 000 до 377;
- в виде экранированного символа. Экранированные символы используются для задания символов в тех позициях, где реальный символ нарушает синтаксис программы. Значения экранированных символов приведены в таблице.

Экранированный символ	Значение
\b	символ возврата
\t	табуляция
\f	перевод страницы
\n	перевод строки
\r	возврат каретки
\"	двойная кавычка
\'	одинарная кавычка
\\	обратная косая черта

Неопределенный литерал `null` представляет неопределенное значение ссылочного типа — пустая ссылка.

Примеры литералов:

```
987_654_321               // десятичный типа int
0765_4321_0123_4567L      // восьмеричный типа long
0xFFFF_CAFE               // шестнадцатеричный типа int
0b0001_0010_0100_1000L    // двоичный типа long
1.                        // десятичный типа double = 1.0
1e0                       // десятичный типа double = 1.0
1d                        // десятичный типа double = 1.0
3.14159F                  // десятичный типа float = π
6.67408e-11f              // десятичный типа float = 6.67408·10^-11
0xFFp4                    // шестнадцатеричный типа double
                          // FF16·2^4 = FF016
true                      // логический
'π'                       // символьный
'\u03c0'                  // символьный = 'π'
"Привет"                  // строковый
```

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`

Области видимости

> *Область видимости объявления* — это часть программы, где объявленная сущность доступна по простому имени.

Все переменные имеют область видимости имени. Обычно область видимости совпадает с блоком, где объявлена переменная (рис. 1).

Правила, по которым определяется область видимости:

- область видимости элемента, указанного в инструкции import – все классы и интерфейсы, объявленные в том же файле;
- область видимости класса или интерфейса верхнего уровня — пакет, к которому принадлежит этот класс или интерфейс;
- область видимости объявленного или наследуемого члена класса или интерфейса — все тело класса или интерфейса, включая вложенные типы;
- область видимости перечисляемой константы — все тело соответствующего перечисляемого типа, а также весь блок switch, использующий данный перечисляемый тип
- область видимости параметра метода, конструктора, цикла или лямбда-выражения — все тело метода, конструктора, цикла или лямбда-выражения;

- область видимости параметра типа элемента — все тело элемента, исключая модификаторы;
- область видимости локального класса или переменной, объявленной в любом блоке — до конца этого блока;
- область видимости параметра обработчика исключения, объявленого в блоке catch – весь блок catch;
- область видимости переменной, объявленной в блоке try с ресурсом — весь блок try.

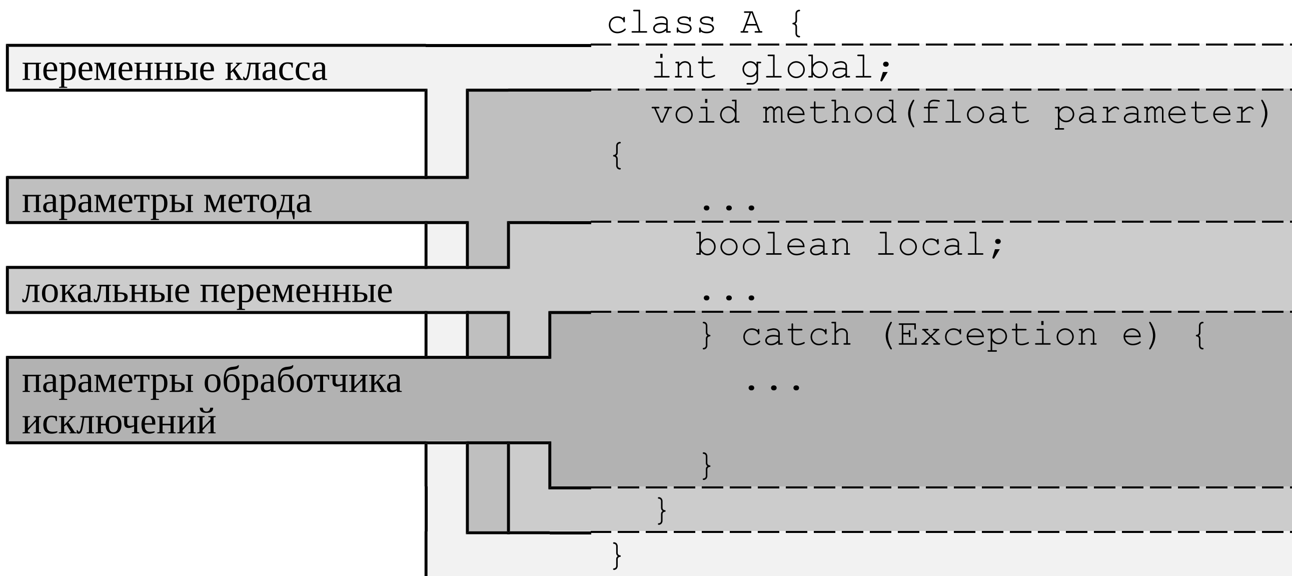


Рисунок 1. Области видимости переменных

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	= += -= *= /= %= >>= <<= >>>= &= ^= |=

стрелка лямбда	`->`

Управляющие конструкции

```
// ветвление

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

```
// цикл с параметром
/*
init - инициализация
expr - условие продолжения
incr - приращение параметра
*/
for (init; expr; incr) {
  op;
}
```

```
// множественный выбор

switch (int|String|enum expr) {
  case const1:
    op1;
  case const2:
    op2;
    break;
  default:
    op3;
}
```

```
// цикл по элементам
/*
coll – любой тип, реализующий
интерфейс Iterable<T> (напри-
мер, коллекция) или массив, со-
держащий элементы типа T
*/

for (T x : coll) {
  op;
}
```

```
// цикл с предусловием

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

```
// цикл с постусловием

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

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

```
// метка для выхода из цикла
label:
  for (...) {
    for (...) {
      op;
      continue label;
    }
  }
```

```
// генерация исключения
throw new Exception();
```

```
// возврат из функции
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:

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

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

@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;
  }
}
```

Стандартная библиотека классов

Пакеты, входящие в JDK 1.8

`java.applet`	Классы для реализации апплетов.
`java.awt`	Классы для реализации графического пользовательского интерфейса.
`java.awt.color`	Классы для раскраски компонентов пользова-тельского 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.annotation`	Классы для поддержки работы с аннотациями
`java.lang.instrument`	Классы для модификации байт-кода
`java.lang.invoke`	Классы для поддержки динамических вызовов
`java.lang.management`	Классы для поддержки мониторинга и управления приложениями.
`java.lang.ref`	Классы, обеспечивающие ряд возможностей по взаимодействию со «сборщиком мусора» вир-туальной 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
`java.nio.channels`	Классы для реализации каналов
`java.nio.charset`	Поддержка кодировок
`java.nio.file`	Новый интерфейс работы с файлами
`java.rmi`	Классы и интерфейсы для обеспечения удален-ного вы-

	зова методов.
java.rmi.activation	API, реализующее возможности по активации RMI-объектов.
java.rmi.dgc	Классы и интерфейсы для реализации распреде-ленной "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.time	Новый интерфейс работы с датами и временем
java.util	Вспомогательные классы, обеспечивающие рабо-ту со структурами данных и форматирование текста с учетом локализации.
java.util.concurrent	Классы, обеспечивающие расширенные возмож-ности многопоточного программирования.
java.util.function	Набор функциональных интерфейсов
java.util.jar	Классы для работы с JAR-архивами.
java.util.logging	Классы и интерфейсы, реализующие журналиро-вание 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.stream	Классы для конвейерных операций с данными
java.util.zip	Классы для обеспечения архивации.
javax.annotation	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 использо-вать 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, преобразо-вания 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 компонен-том JColorChooser.
`javax.swing.event`	Классы событий, создаваемых компонентами Swing.
`javax.swing.filechooser`	Классы и интерфейсы, используемые компонен-том JFileChooser.
`javax.swing.plaf`	Классы и интерфейсы, реализующие возмож-ности 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 редактиру-емые 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.

Пакет java.lang

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

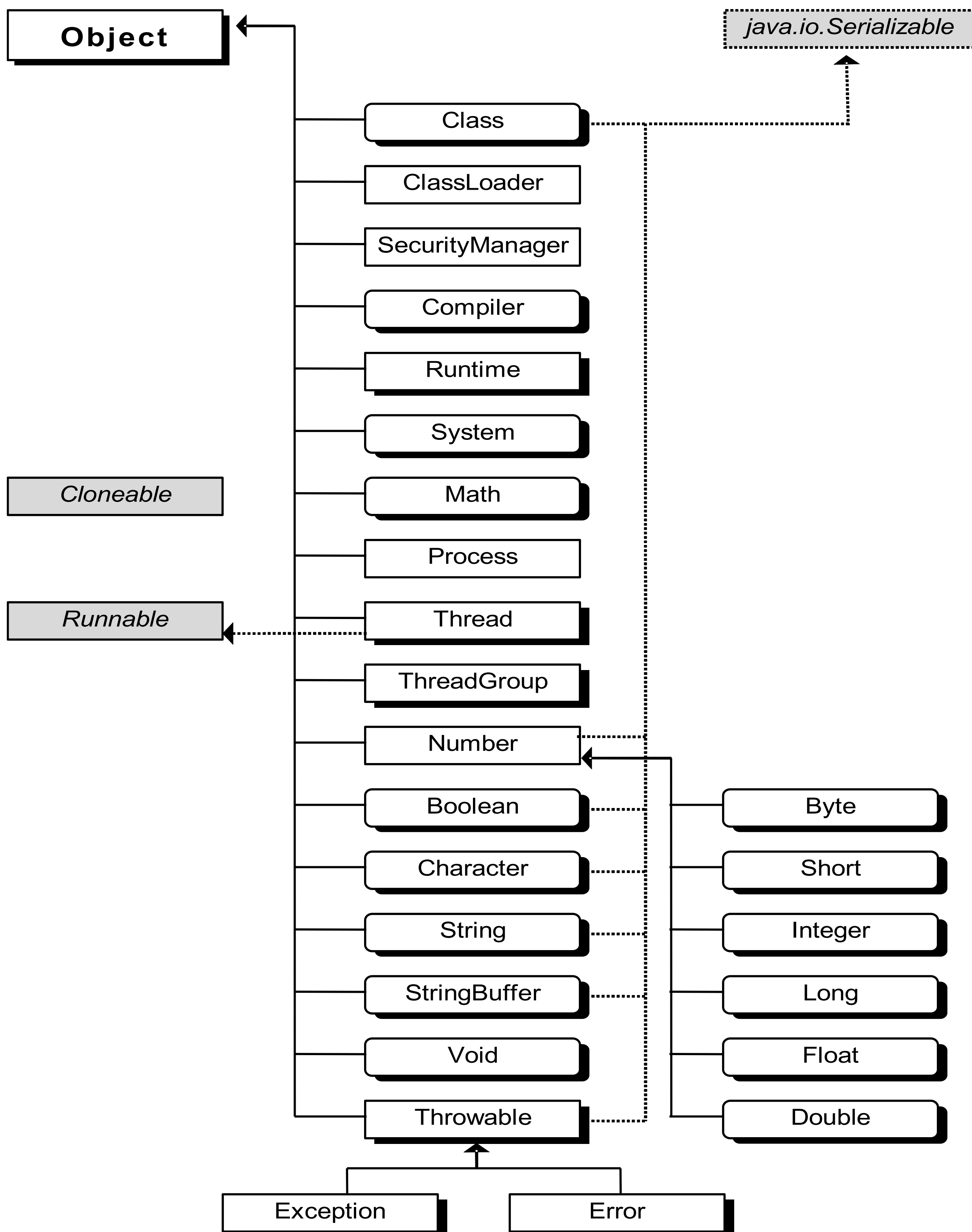


Рисунок 2. Пакет java.lang

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 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

Классы StringBuffer и StringBuilder

Используются для представления изменяемых строк, содержат идентичные методы, но StringBuilder работает несколько быстрее, а StringBuffer может использоваться для работы в многопоточных приложениях.

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 java.util.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());
}
```

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.

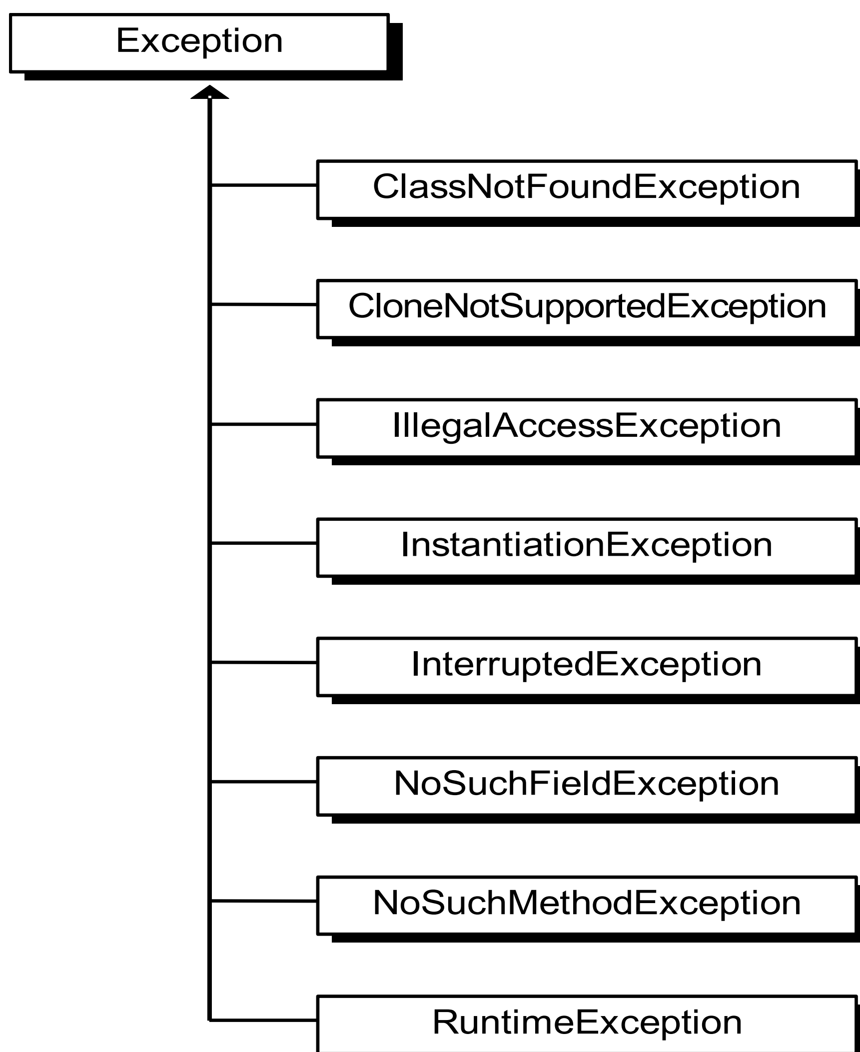


Рисунок 3. Исключения пакета java.lang

Обработка исключений производится с помощью блока try-catch-finally.

Класс RuntimeException

Данные исключения описывают исключения среды исполнения, которые могут возникать в любом месте программы, и которые не нужно объявлять в операторе `throws`. В отличие от остальных исключений, потомки класса `RuntimeException`, являются неконтролируемыми, так как компилятор не требует их обработки.

Основные классы неконтролируемых исключений приведены на рис. 4.

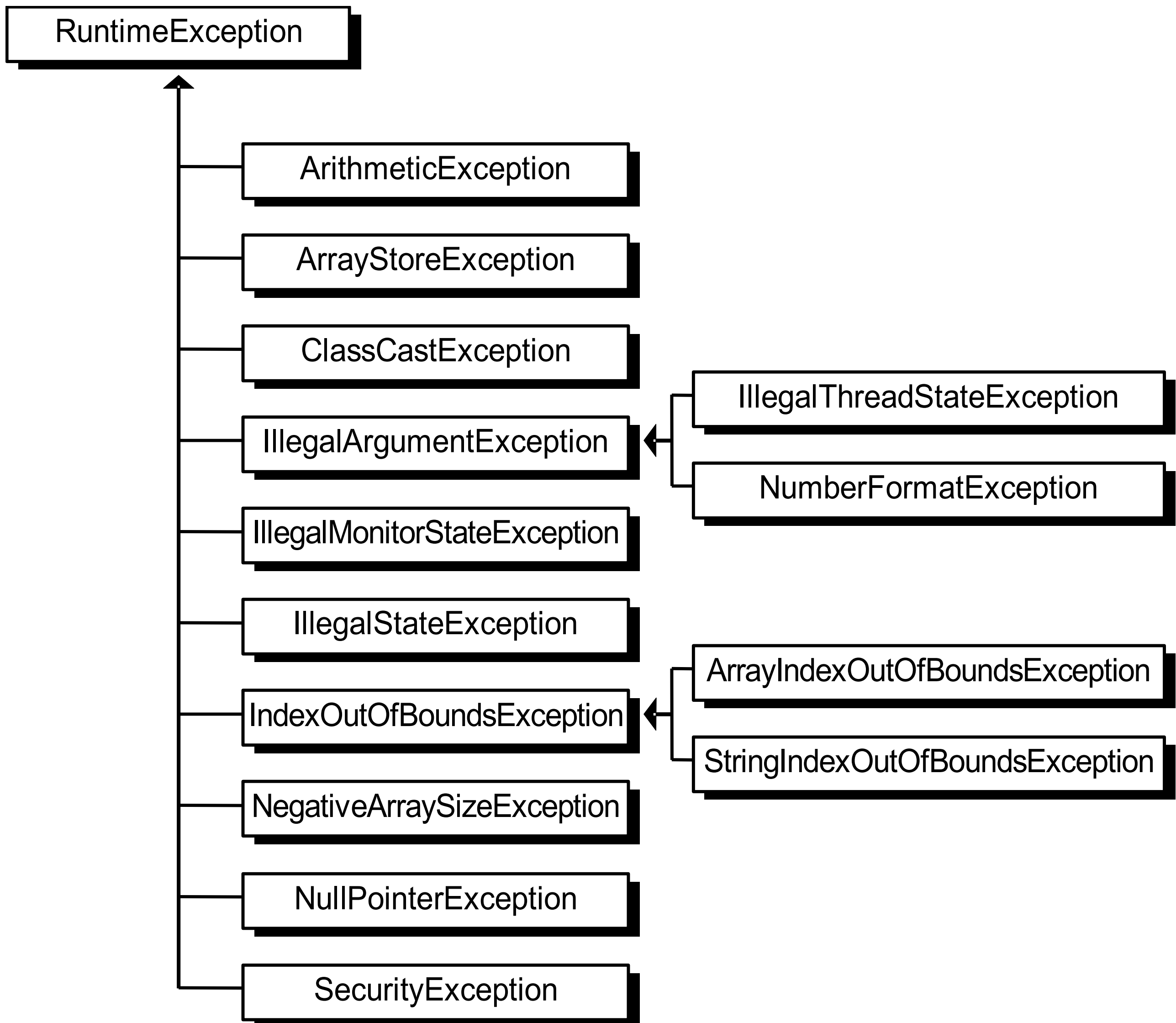


Рисунок 4. Неконтролируемые исключения пакета `java.lang`

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.

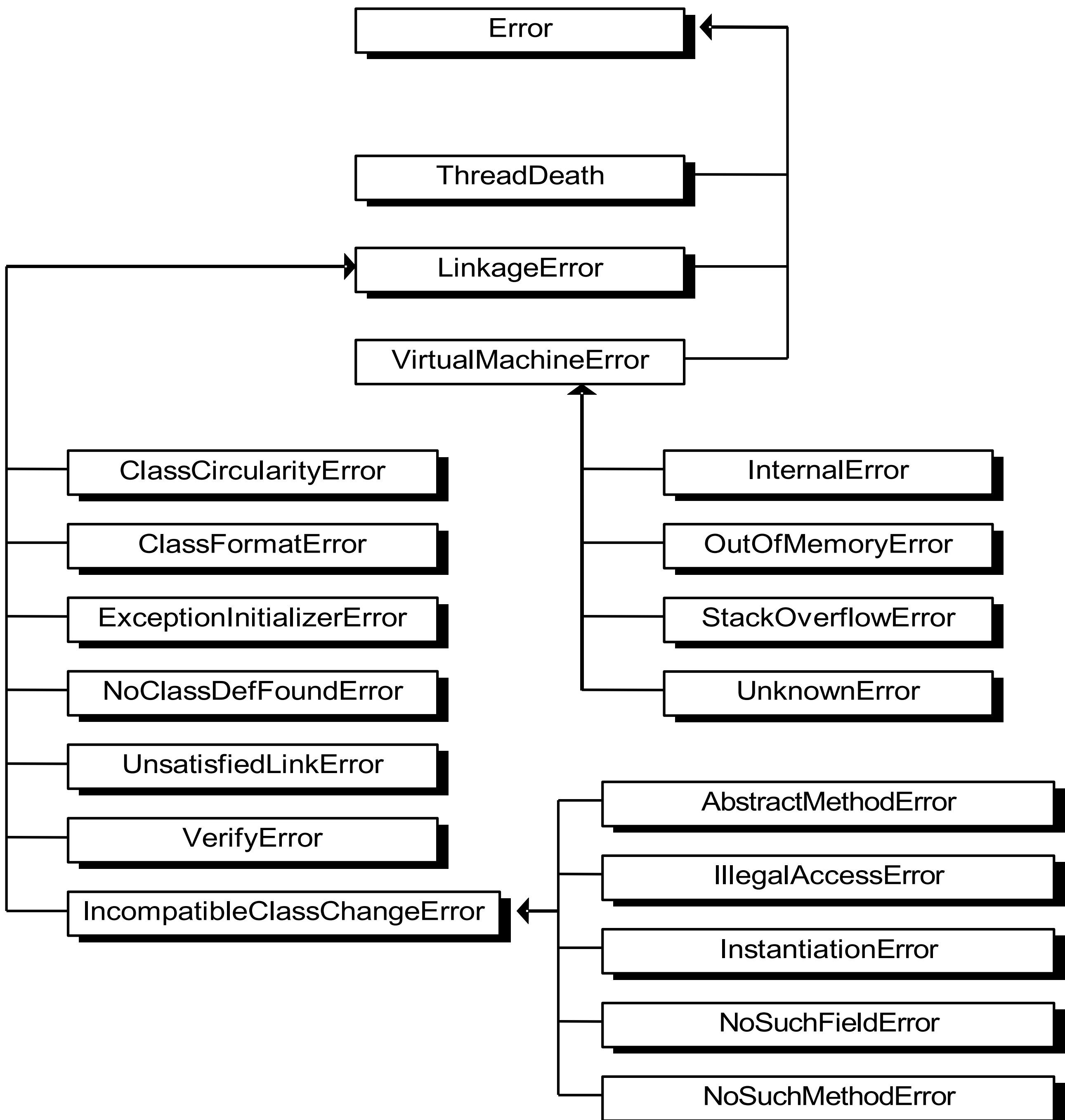


Рисунок 5. Ошибки пакета java.lang

Множественная обработка исключений

В Java 7 и выше единственный блок `catch` может обрабатывать более одного исключения разных типов. Это позволяет уменьшить количество повторяющегося кода и снижает вероятность перехватить слишком общее исключение.

Следующий код содержит повторяющиеся фрагменты в каждом блоке `catch`:

```
catch (IOException ex) {
     logger.log(ex);
     throw ex;
catch (SQLException ex) {
     logger.log(ex);
     throw ex;
}
```

В предыдущих версиях Java было сложно создать метод, который бы позволил не дублировать код, так как переменная ex разного типа.

Ниже приведен пример для Java 7 и выше, решающий проблемы дубликации кода:

```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

В описании блока catch указаны типы исключений, которые блок может обрабатывать. Названия типов разделены вертикальным разделителем «|»

При этом следует учитывать, что если в блоке catch указано более одного типа исключений, то параметр блока catch будет являться final переменной. В примере выше параметр ex будет final и внутри блока catch ему нельзя будет присваивать другие значения.

Проброс исключений более узких типов

Компилятор Java 7 осуществляет более точный анализ преобразовываемых исключений. Это позволяет указывать более узкие виды исключений в блоке `throws` при описании метода.

Рассмотрим пример:

```
static class FirstException extends Exception { }
  static class SecondException extends Exception { }

  public void rethrowException(String exceptionName) throws
Exception {
    try {
      if (exceptionName.equals("First")) {
        throw new FirstException();
      } else {
        throw new SecondException();
      }
    } catch (Exception e) {
      throw e;
    }
  }
```

В данном примере блок try может выбрасывать исключения типов `FirstException` и `SecondException`. Предположим, что требуется указать эти

исключения в блоке `throws` для дальнейшего проброса. До Java 7 этого было нельзя сделать, так как параметр блока `catch`, `e`, имеет тип `Exception` и пробрасываемое исключение будет типа `Exception`. То есть, при объявлении метода можно было указать только этот тип, `Exception`.

В Java 7 компилятор определяет, какое именно исключение будет выброшено из блока `try`. В данном примере этот будут только исключения типов `FirstException` и `SecondException`. Даже если параметр блок `catch`, `e`, будет объявлен более общий типом исключений (`Exception`), компилятор определит, является ли пробрасываемое исключение экземпляром класса `FirstException` или `SecondException`.

```
public void rethrowException(String exceptionName)
  throws FirstException, SecondException {
    try {
      // ...
    }
    catch (Exception e) {
      throw e;
    }
  }
```

Проверка типа исключения не происходит, если параметру `e` внутри блока `catch` будут присваиваться какие-то другие значения.

Таким образом, когда объявляется одно или более исключений в блоке `catch` и затем эти исключения пробрасываются, компилятор проверяет удовлетворяет ли тип пробрасываемых исключений следующим условиям:

- блок `try` должен быть способен их выбросить;
- нет больше никаких блоков `catch`, которые могли бы их перехватить;
- исключение является предком или потомком любого исключения, объявленного в параметрах `catch`

Компилятор Java 7 позволит указать исключения типов `FirstException` или `SecondException` при в блоке `throws` при объявлении метода так как вы можете пробросить любое исключение, если оно является предком объявленных в `throws`.

В более ранних версиях Java нельзя было выбросить исключение которое является предком исключений указанных в параметрах блока `catch`. В этом случае на выражение `throw e;` в коде компилятор бы сгенерировал ошибку «`unreported exception Exception; must be caught or declared to be thrown`» (незарегистрированное исключение `Exception`; должно быть отловлено или объявлено как пробрасываемое)

Выражение try-with-resources.

Выражение try-with-resources - это такое выражение, которое объявляет блок `try` с одним и более ресурсами (источниками). Ресурс, это объект, который может быть закрыт, после того как программа закончила с ним работу. Выражение try-with-resources обеспечивает то, что каждый объявленный в нем источник будет закрыт в конце блока. Любой объект, который реализует интерфейс

java.lang.AutoCloseable, включает в себя объекты, каждый из которых реализует интерфейс java.io.Closeable, может являться ресурсом.

Пример ниже считывает первую строку из файла. Для того, чтобы прочитать данные из файла она использует экземпляр класса BufferedReader. BufferedReader это ресурс, который может быть закрыт после того как программа закончит с ним работу:

```
static String readFirstLineFromFile(String path) throws
IOException {
    try (BufferedReader br =
                   new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

В этом примере ресурс, объявленный выражением try-with-resources — это BufferedReader. Выражение с объявлением ресурса находится в фигурных скобках непосредственно после ключевого слова try. Класс BufferedReader, начиная с Java 7, реализует интерфейс java.lang.AutoCloseable. Так как экземпляр BufferedReader объявлен в выражении try-with-resources, он будет закрыт вне зависимости от того нормально ли завершилось выполнение выражения try или нет (например метод return br.readLine() выбросит исключение).

Ранее в таких ситуация было принято применять блок finally — чтобы закрыть ресурс вне зависимости от результата его использования. Пример ниже это иллюстрирует:

```
static String readFirstLineFromFileWithFinallyBlock(String path)
                                            throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) br.close();
    }
}
```

Тем не менее, если оба метода br.readLine() и br.close() выбросят исключения, то метод readFirstLineFromFileWithFinallyBlock выбросит то исключение, которое сгенерировал блок finally. То есть, исключение из блока try будет подавлено. В отличие от этого, при использовании try-with-resources метод readFirstLineFromFile выбросит исключение из блока try, при этом будет подавлено внутреннее исключение блока try-with-resources.

В блоке try-with-resources можно объявлять несколько ресурсов. Следущий пример получает имена файлов из zip-архива и записывает их в создаваемый текстовый файл.

```
public static void writeToFileZipFileContents(String zipFileName,
                                           String outputFileName)
                                           throws
```

```
java.io.IOException {

    java.nio.charset.Charset charset =
         java.nio.charset.StandardCharsets.US_ASCII;
    java.nio.file.Path outputFilePath =
         java.nio.file.Paths.get(outputFileName);


    try (
        java.util.zip.ZipFile zf =
             new java.util.zip.ZipFile(zipFileName);
        java.io.BufferedWriter writer =
            java.nio.file.Files.newBufferedWriter(outputFilePath,
                                                 charset)
    ) {
      for (java.util.Enumeration entries = zf.entries();
                              entries.hasMoreElements();) {
          String newLine = System.getProperty("line.separator");
          String zipEntryName =
    ((java.util.zip.ZipEntry)entries.nextElement()).getName() +
             newLine;
           writer.write(zipEntryName, 0, zipEntryName.length());
        }
    }
}
```

В данном примере выражение try-with-resources содержит два объявления, разделенные точкой с запятой (`;`). Методы `close()` классов `ZipFile` и `BufferedWriter` будут вызываться автоматически в определенном порядке после того как, нормально или с исключением, завершится выполнение блока кода, который следовал сразу за объявлением. Порядок вызова метода является обратным относительно создания соответствующих ресурсов.

У выражения try-with-resources могут быть блоки `catch` и `finally`, так же, как и у обычного `try`. Любой из блоков `catch` или `finally` будут вызываться только после того, как будет выполнено закрытие всех объявленных ресурсов.

Подавленные исключения

Исключение может быть выброшено из блока ассоциированного с выражением try-with-resources. В примере выше, одно исключение может быть выброшено из блока `try` и одно или два исключения из выражения try-with-resources, при попытке закрытия ресурсов. Если одновременно выброшены исключения из блока `try` и из выражения try-with-resources, то исключения из try-with-resources будут подавлены. Метод `writeToFileZipFileContents` выбросит только исключение из блока `try`. Подавленные исключения можно получить из выброшенного исключения с помощью метода `Throwable.getSuppressed`.

Пакет java.util

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

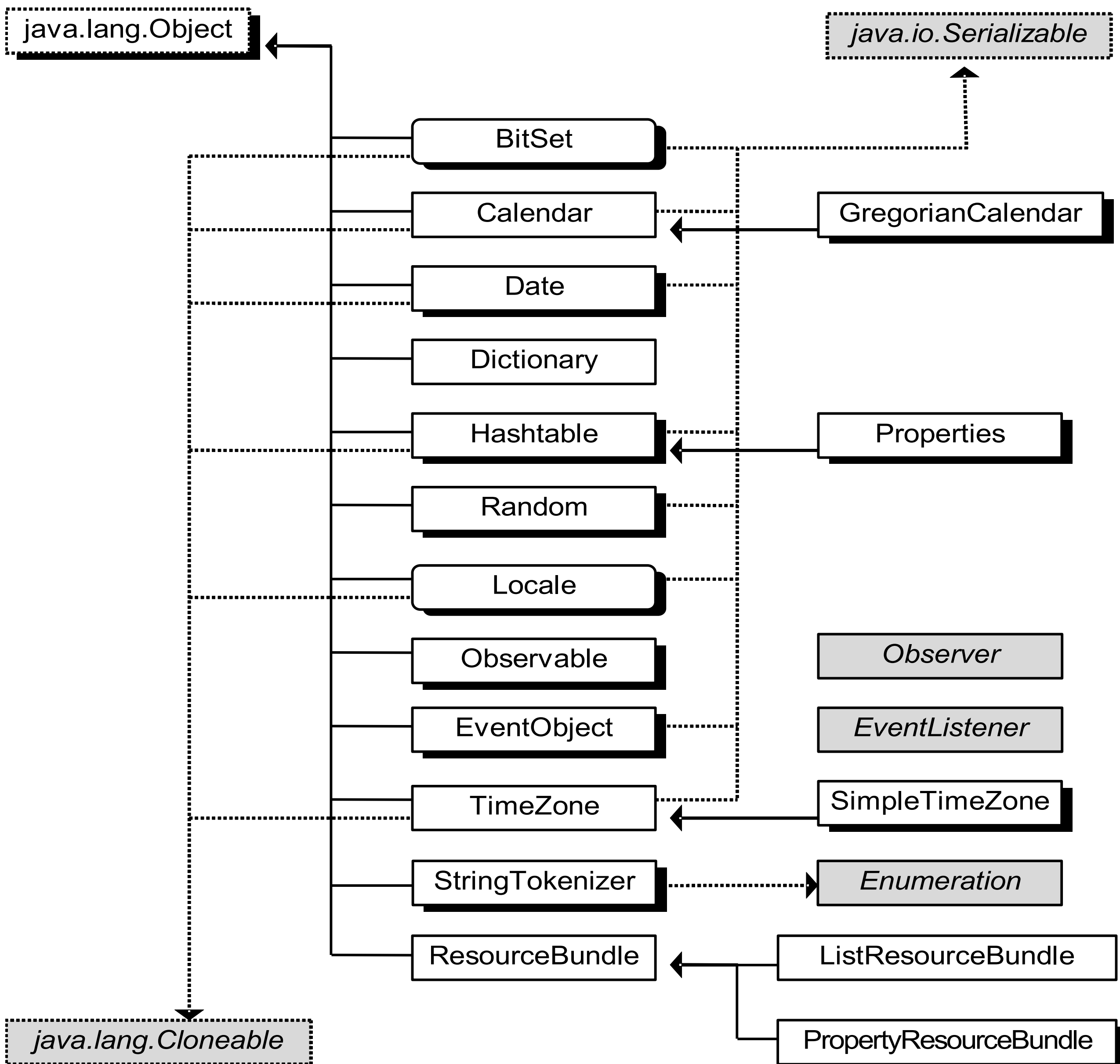


Рисунок 6. Классы и интерфейсы пакета `java.util`

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

Использование легковесных процессов

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

Потоки используются при решении многих задач:

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

Другим способом реализации многозадачности являются процессы. Основное отличие процессов от потоков состоит в том, что процесс обладает собственным адресным пространством и обменивается данными с другими процессами при помощи межпроцессного взаимодействия. В то время как потоки одного процесса имеют общее адресное пространство и могут обращаться к одной области памяти, используя для обмена данными общие переменные. Переключения контекста потока происходит намного быстрее, чем для процесса, поэтому создание и уничтожение потоков требуют меньше времени и системных ресурсов.

Реализация потока

Java предусматривает две возможности реализации потоков:

- класс реализует интерфейс `java.lang.Runnable` с определением метода `run()`, затем экземпляр этого класса передается в конструктор класса `Thread`;

```
public class MyThread1 implements Runnable {
     public void run() {
          // тело потока
     }
}
...
Thread t = new Thread(new MyThread1);
t.start();
```

- класс строится как потомок класса `java.lang.Thread` с переопределением метода `run()`, затем создается экземпляр этого класса.

```
public class MyThread2 extends Thread {
     @Override public void run() {
          // тело потока
     }
}
...
MyThread2 t = new ldap.devMyThread2();
```

```
t.start();
```

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

Запуск потока осуществляется с помощью метода start() класса Thread. При этом подготавливаются необходимые для запуска ресурсы, создается новый поток и начинает выполняться его метод run().

Состояние потока

Состояния потока с точки зрения виртуальной машины определены во вложенном классе Thread.State:

- NEW (Новый) — поток создан, но еще не запущен методом start().
- RUNNABLE (Выполняемый) — в данное состояние поток переходит после вызова метода start(), а также после возврата из других состояний. При этом потоками управляет диспетчер, который распределяет процессорное время между выполняемыми потоками, выделяя каждому небольшой квант времени. Количество одновременно выполняющихся потоков зависит от количества процессоров и количества ядер в процессорах. Если поток находится в состоянии RUNNABLE, он либо реально выполняется процессором, либо ожидает предоставления ему кванта процессорного времени. Виртуальная машина не различает эти 2 подсостояния.
- BLOCKED (Заблокированный) — поток заблокирован ожиданием монитора.
- WAITING (Ожидающий) — поток ждет выполнения другим потоком определенного действия.
- TIMED_WAITING (Временно ожидающий) — поток ждет выполнения другим потоком определенного действия в течение определенного времени, по истечении которого поток прекращает ожидание.
- TERMINATED (Завершившийся) — метод run() завершил работу.

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). Потоки с более высоким приоритетом будут первыми выбираться для выполнения.

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 ра-

ботой. С группой потоков возможны те же основные операции, что и с простым потоком:

- запуск;
- останов;
- установка приоритетов;
- и так далее.

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 устанавливают.

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 информацию о том, является ли *текущий* поток остановленным (статический метод);
- `isInterrupted()` возвращает информацию о том, остановлен ли *выбранный* поток (нестатический метод);
- `isAlive()` возвращает информацию о том, жив ли поток;
- `isDaemon()` возвращает информацию о том, является поток демоном.

Вторая группа:

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

Взаимодействие и синхронизация потоков

Поток может приостановить свое выполнение на определенное время с помощью метода `Thread.sleep(long ms)`. При этом поток переходит в состояние `TIMED_WAITING`, и дает возможность поработать другим потокам.

С помощью метода `join()` можно заставить поток ждать, когда другой поток завершит свою работу.

Метод `interrupt()` позволяет послать потоку запрос на прерывание путем установки флажка. Во время работы поток может периодически проверять состояние флажка с помощью метода `isInterrupted()` и при получении запроса на прерывание, предпринять необходимые действия (обычно это завершение работы). Методы `sleep()`, `join()` и другие при получении запроса на прерывание выбрасывают `InterruptedException`, которое можно перехватить и обработать.

Если два потока имеют доступ к одному и тому же разделяемому ресурсу, то при работе этих потоков возможно наступление состояния гонок (*race condition*), когда один из потоков считал данные, изменил их, но до того, как он успел записать измененные данные, другой поток прочитал их, и также изменил. Когда оба потока выполнят операцию записи, данные станут недостоверными. Для исключения подобных ситуаций, производится синхронизация. Тот участок кода, где происходит считывание и запись данных, помечается ключевым словом `synchronized`. Есть два варианта использования `synchronized` – синхронизированный блок и синхронизированный метод.

Любой объект в Java имеет монитор, использующийся для синхронизации. При выполнении синхронизированного блока будет использоваться монитор объекта, указанного в качестве параметра блока.

```
synchronized (obj) {
    a = a + x;
}
```

При входе в синхронизированный блок одного из потоков, данный поток захватывает монитор объекта `obj`. Пока монитор занят, другие потоки не могут войти в синхронизированный блок, переходя в состояние `BLOCKED`. При выходе из блока, первый поток освобождает монитор, разрешая тем самым заблокированным потокам попытаться захватить монитор и получить разрешение на выполнение синхронизированного блока.

Важное замечание — поток может захватывать монитор повторно, если он уже владеет этим монитором.

Если несколько синхронизированных блоков используют один и тот же объект для синхронизации, то все эти блоки становятся недоступными для выполнения при захвате одним из потоков монитора данного объекта. При использовании разных объектов блоки могут выполняться одновременно.

Синхронизированные методы — это методы с модификатором `synchronized`. Они используют в качестве объекта синхронизации текущий объект (`this`). Если синхронизированный метод — статический, то блокироваться будет объект класса `Class`, соответствующий данному объекту.

Для того, чтобы потоки могли извещать друг друга о наступлении некоторого события, например, о том, что данные подготовлены для считывания, можно использовать методы `wait()`, `notify()`, `notifyAll()` класса `Object`.

Например, есть два метода: `put` и `get`, один из которых устанавливает значение переменной, другой читает его. Необходимо синхронизировать методы так, чтобы один из потоков мог устанавливать значение после того, как предыдущее будет прочитано, а второй поток мог читать значение после того, как оно будет установлено другим потоком.

```
class A {
  boolean flag = false;
  int value;
```

```
  synchronized void put(int i) {
    while(flag) { wait(); }
    flag = true; value = i;
    notifyAll();
  }
  synchronized int get() {
    while(!flag) { wait(); }
    flag = false;
    notifyAll();
    return value;
  }
}
```

Методы `wait()`, `notify()` и `notifyAll()` работают с использованием внутреннего монитора объекта. При вызове этих методов поток должен обладать монитором, поэтому эти методы должны всегда располагаться внутри синхронизированных блоков или методов.

Метод `wait()` помещает поток в список ожидания объекта, после чего освобождает монитор и переходит в состояние `WAITING` или `TIMED_WAITING`. При этом метод `wait()` остается в незавершенном состоянии. Другие потоки могут захватить свободный монитор и начать выполнять синхронизированный блок. Как только некоторый поток совершил действие, которое ожидают другие потоки, он вызывает метод `notify()` или `notifyAll()` и выходит из синхронизированного блока, освобождая монитор. Методы `notify()` и `notifyAll()` выводят из состояния ожидания либо один, либо все потоки, которые находились в списке ожидания данного объекта. Эти потоки пытаются захватить монитор, получив который, они могут завершить выполнение метода `wait()` и продолжить работу.

Модификатор volatile

Модификатор `volatile` применяется для переменных и означает, что:

1. Переменная, объявленная `volatile`, не кэшируется потоком (что для обычных переменных может происходить для оптимизации), а всегда читается или записывается напрямую в память.
2. При операциях чтения-записи из нескольких потоков гарантируется, что операция записи для `volatile`-переменной будет завершена до операции чтения.
3. Операции чтения-записи для `volatile`-переменной всегда атомарны (даже для типов `long` и `double`)

Обобщенное программирование

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

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

Шаблоны

Шаблон (generic) – описание класса, метода или атрибута без использования конкретного типа данных.

Без шаблонов	С шаблонами

Объявление:
List l = new LinkedList();
l.add(new Integer(0));
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));
	l.add(new
	Double(1.1);//compilation error
	
	Использование:
	for (…) {
	   Integer x =
	l.iterator().next();
	}

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

Пример показывает, что 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.

Например, код

```
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

Обобщенный тип (generic type) – это описание класса с использованием формальных параметров типа.

Параметризованный тип (parameterized type) — 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

Метод с использованием формального описания типа называется шаблонным методом (generic method).

Признаком того, что метод является шаблонным служит указание типа данных, с которым работает 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

Формальный параметр типа (type parameter) — 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 позволяет задать возможные границы подстановки конкретных типов данных для получения параметризованного типа.

Например, `c extends Number` разрешает использовать в качестве формального параметра только потомков класса `Number`.

Шаблоны с групповой подстановкой

Wildcards (дословно Джокер) или групповая подстановка – синтаксические конструкции с использованием символа '?', используемые для замены в шаблонах конкретного класса множеством возможных классов.

Пример:

```
/*
```

```
 * 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 множеством объектов.

Иерархия коллекций представлена на рис. 7.

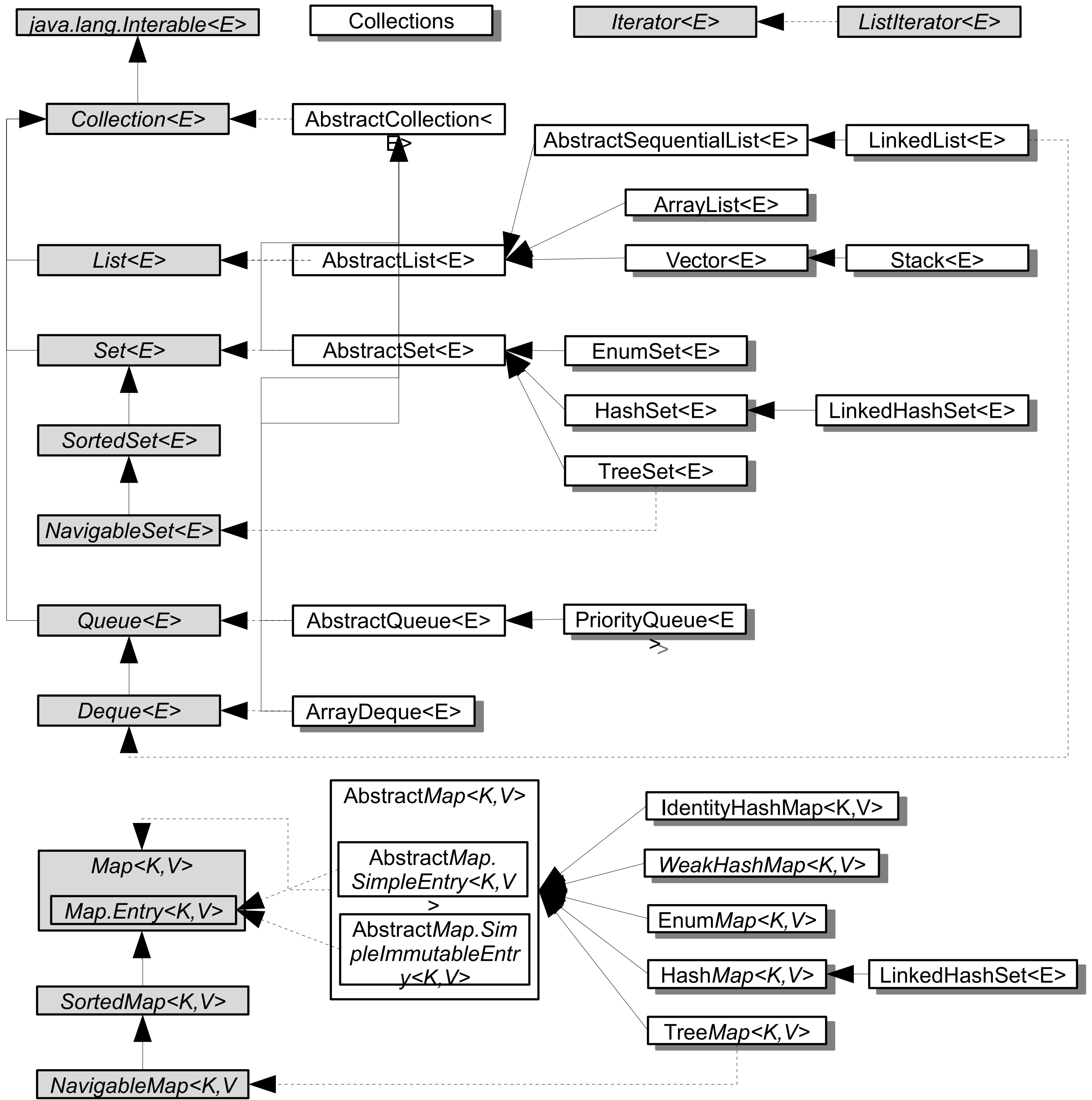


Рисунок 7. Коллекции

Интерфейс Iterator

Iterator – интерфейс, позволяющий выполнять обход элементов коллекции. 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.

Интерфейс 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.

Методы Collection могут быть не реализованы в конкретной реализации той или иной коллекции. В этом случае они обязаны генерировать исключение UnsupportedOperationException. Любая коллекция должна возвращать итератор, и соответственно, позволять перечислять 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)
NavigableSet	SortedSet, расширенный методами кратчайшего доступа к искомому элементу. В NavigableSet элементы могут быть доступны в порядке увеличения и уменьшения
NavigableMap	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 другие, 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 раскрашенных деревьев.
LinkedHashSet	Реализация множества в виде хеш таблицы и двусвязанного 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 двусвязанного списка с заданным порядком элементов.

В дополнение к общим реализациям существуют прародители коллекций — классы Vector и Hashtable, которые были обновлены 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.equals(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.

Пакет 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.

Класс 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 так далее.
`synchronizedCollection (Collection)`	Создает потоко-безопасную копию коллекции. Существуют отдельные методы для `Set`, `List`, `Map`, и так далее.
`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 так далее.
`<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 Queue-представление из `Deque`.

Лямбда-выражения

Язык Java реализует императивную парадигму программирования.

> *Императивное программирование* — это парадигма программирования, которая описывает процесс вычисления в виде инструкций, изменяющих состояние данных.

В 8 версии Java разработчики добавили в синтаксис языка некоторые элементы функционального программирования.

> *Функциональное программирование* — это парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании).

Иными словами, если в программе на императивном языке программирования результат вычисления функции может определяться как значениями ее аргументов, так и текущим состоянием программы (совокупностью значений переменных), то в программе на функциональном языке результат вычисления одной и той же функции с одним и тем же набором аргументов всегда будет одинаков.

Следующий пример иллюстрирует отличия процедурного стиля программирования от функционального.

```
// Процедурный стиль – внешняя переменная
// хранит состояние
int a = 0;

void increment1() {
    a++;
}

// Функциональный стиль – состояние не сохраняется
int increment2(int a) {
    return a++;
}
```

Ключевые особенности функциональных языков программирования:

- Все функциональные языки программирования базируются на концепции *лямбда-исчисления* (λ-исчисления) – формальной системы, в основе которой лежит понятие анонимной функции. Эта система была предложена американским математиком А. Чёрчем в 1932 году; она способна определить в своих терминах любую языковую конструкцию или алгоритм.
- *Функции высших порядков* – в функциональных языках программирования функции могут принимать в качестве аргументов и возвращать другие функции.
- Использование *чистых функций*. Чистые функции – это функции, которые зависят только от своих аргументов и возвращают только свой результат.

- Вместо циклов обычно используется *рекурсия* – функция вызывает саму себя.

Так как Java является императивным объектно-ориентированным языком, все функциональные нововведения в ней являются “косметическими”. Они представляют собой так называемый “синтаксический сахар”, то есть позволяют писать код более компактно, но никак не влияют на основы самого языка программирования.

Концепция лямбда-выражений в Java возникла по причине частого использования в программах так называемых функциональных интерфейсов.

> *Функциональный интерфейс* – это интерфейс, в котором определен один (и только один) абстрактный метод.

Важное примечание – если в интерфейсе объявлены методы, существующие в классе `Object` (например, `toString` или `equals`), то такие методы при подсчете количества абстрактных методов не учитываются. К примеру, интерфейс `java.util.Comparator` является функциональным, несмотря на то, что в нём описаны два абстрактных метода – `compare` и `equals`: метод `equals` есть в классе `Object`, поэтому учитывается только метод `compare`.

Другими примерами функциональных интерфейсов являются `Runnable`, `Callable` а также разнообразные слушатели событий в AWT, Swing и JavaFX.

В Java 8 появилась возможность в явном виде объявить интерфейс функциональным – для этого нужно пометить его аннотацией `@FunctionalInterface`. В этом случае компилятор проверит, действительно ли этот интерфейс является функциональным, и, если нет (к примеру, если в нем описан более, чем один метод), выдаст ошибку.

Нововведение Java 8 состоит в том, что теперь вместо функциональных интерфейсов появилась возможность использовать лямбда-выражения.

> *Лямбда-выражение* – это блок кода с параметрами, который может быть преобразован в функциональный интерфейс.

Рассмотрим пример. Пусть нам необходимо отсортировать студентов по их возрасту. В Java 7 код, реализующий это, будет выглядеть примерно так:

```
public class Student {
  ...
  Date birthday;

  public int getAge() {
    // Вычисляем возраст из даты рождения
  }

  public Date getBirthday {
    return birthday;
  }
}
```

```
// Реализация функционального интерфейса Comparator
class StudentAgeComparator
  implements Comparator<Student> {

  // Сравниваем студентов по возрасту
  public int compare(Student a, Student b) {
    return a.getBirthday().compareTo(b.getBirthday());
  }
}

// Сортируем массив с помощью созданного компаратора
Arrays.sort(strs, new StudentAgeComparator());
```

В Java 8 аналогичный код может выглядеть существенно лаконичнее, если воспользоваться лямбда-выражениями:

```
Student[] students = ...; // Сортируемый массив
// Компаратор создаётся с помощью лямбда-выражения
Comparator<Student> comp =
  (Student a, Student b) -> {
    a.getBirthday().compareTo(b.getBirthday());
  };
Arrays.sort(strs, comp);
```

Таким образом, вместо создания отдельного класса, реализующего интерфейс `Comparator`, во втором примере используется анонимная функция. Оператор "->" отделяет список аргументов анонимной функции от ее тела.

Реализовав код компаратора прямо в вызове метода сортировки, приведенный выше пример можно еще упростить:

```
// Лямбда-выражение объявляется прямо в вызове
// метода сортировки
Arrays.sort(students, (Student a, Student b) ->
  { a.getBirthday().compareTo(b.getBirthday()) });
```

Однако, и это еще не все. Если тело единственного метода реализации функционального интерфейса состоит только из одного выражения, то фигурные скобки можно опустить. Помимо этого, если компилятор может из контекста вывести типы параметров лямбда-выражения, их тоже можно не указывать. Таким образом, код в нашем примере станет еще проще:

```
// Не указываем фигурные скобки и типы параметров
// лямбда-выражения
Arrays.sort(students,
  (a, b) -> a.getBirthday().compareTo(b.getBirthday()));
```

Важно учитывать, что тело лямбда-выражения не может создавать проверяемые исключения, за исключением тех, которые указаны в заголовке метода функционального интерфейса.

```
// Метод run() не пробрасывает InterruptedException -
// код не скомпилируется
```

```
Runnable sleepingRunner = () -> {
    Thread.sleep(1000); // Throws InterruptedException
};
```

Если немного модифицировать исходный класс в приведенном выше примере с сортировкой массива, то объем кода можно сократить еще больше.

```
public class Student {
  ...
  Date birthday;

  public int getAge() {
    // Вычисляем возраст из даты рождения
  }

  public Date getBirthday {
    return birthday;
  }

  // Добавим метод сравнения по возрасту
  // прямо в классе Student
  public static int compareByAge(Student a, Student b) {
    return a.birthday.compareTo(b.birthday);
  }
}
```

В этом случае лямбда-выражение в функции сортировки можно реализовать так:

```
Arrays.sort(students,
  (a, b) -> Student.compareByAge(a, b));
```

В Java 8, в случае, если лямбда выражение вызывает определенный в каком-либо классе метод с передачей ему своих параметров, и больше ничего не делает, можно вместо него просто указать ссылку на этот метод. Делается это с помощью нового оператора “`::`”:

```
// Вместо лямбда-выражения просто задаём ссылку
// на метод сортировки.
// Компилятор сам сгенерирует класс-компаратор
// на основании этой ссылки.
Arrays.sort(strs, Student::compareByAge);
```

Лямбда-выражение в этом случае будет сгенерировано компилятором автоматически на основании функционального интерфейса и сигнатуры метода `compareByAge`.

Ссылаться с помощью оператора “`::`” можно как на методы экземпляра, так и на статические методы. Помимо этого, можно ссылаться на конструкторы – это позволяет упростить инициализацию массивов объектов:

```
List<String> strs = ...; // Список надписей на кнопках
// Преобразуем в java.util.Stream
Stream<Button> stream = strs.stream();
```

```
// Инициализируем каждую кнопку с помощью ссылки
// на конструктор
Button[] buttons = stream.toArray(Button[]::new);
```

Как и анонимные классы, лямбда-выражения могут обращаться к локальным переменным (и параметрам) функции, в теле которой они объявлены. Таким образом, приведенный ниже код является корректным:

```
public static void repeatText(String text, int count) {
    Runnable r = () -> {
        // Переменные text и count можно использовать
        // внутри лямбда-выражения
        for (int i = 0; i < count; i++) {
            System.out.println(text);
            Thread.yield();
        }
    };
    new Thread(r).start();
}
```

Единственное ограничение – такие “захваченные” переменные внутри лямбда-выражения не должны изменяться (модифицироваться, повторно присваиваться, и так далее). Например, приведенный ниже пример не скомпилируется:

```
public static void repeatText(String text, int count) {
    Runnable r = () -> {
        while (count > 0) {
            count--; // Так делать нельзя
            System.out.println(text);
            Thread.yield();
       }
    };
    new Thread(r).start();
}
```

При этом, если “захваченная” переменная – объектная, то никто не запрещает менять ее состояние из лямбда-выражения, вызывая ее методы (правда, это может быть небезопасно в многопоточных приложениях:

```
public static void fight(int count, Pokemon fighter) {
    Runnable r = () -> {
        for (int i = 0; i < count; i++) {
            fighter.fight(); // Всё корректно
            Thread.yield();
        }
    };
    new Thread(r).start();
}
```

Пакет java.util.concurrent

Рассмотренные в разделе "Использование легковесных процессов" возможности параллельной обработки данных были низкоуровневыми. Начиная с версии 5 в Java появился пакет, содержащий классы для расширения возможностей параллельных приложений, и более удобных средств работы с ними. Рассмотрим подробнее классы, входящие в пакет `java.util.concurrent`. Они делятся на 6 групп.

Исполнители (Executors)

Интерфейс `Callable<V>` предоставляет функциональность, аналогичную `Runnable`, но, в отличие от `Runnable`, имеет возвращаемое значение и может выбрасывать исключение. Содержит метод `V call()`.

Интерфейс `Future<V>` позволяет получить результат работы задачи в будущем. Содержит методы:

- `V get()` – блокирует текущий поток до завершения операции и возвращает значение
- `V get(long, TimeUnit)` – блокирует поток до завершения операции или таймаута
- `boolean cancel(boolean)` – пытается отменить задачу. Если задача еще не началась, возвращает `true` и успешно ее отменяет. Если задача уже завершилась, возвращает `false`. Если задача выполняется, то при параметре `true` пытается ее прервать, при параметре `false` разрешает ей завершиться самостоятельно.
- `boolean isDone()` – возвращает `true`, если задача завершилась любым способом (нормально, в результате отмены или исключения).
- `boolean isCancelled()` – возвращает `true`, если задача была отменена.

Интерфейс `Delayed` –– позволяет задать время до начала задачи. Содержит метод `long getDelay(TimeUnit)`.

Интерфейс `RunnableFuture` – объединение `Runnable` и `Future`. Позволяет запустить задачу и получить ее результат, когда он будет получен.

Интерфейс `ScheduledFuture` – объединение `Delayed` и `Future`.

Интерфейс `RunnableScheduledFuture` – объединение `Runnable` и `ScheduledFuture`. Позволяет запустить задачу в заданное время и получить ее результат. Дополнительно содержит метод `boolean isPeriodic()`, позволяющий задавать повторяющиеся задачи.

Базовый интерфейс `Executor` выполняет предоставленную ему задачу, реализующую интерфейс `Runnable`. Он обеспечивает разделение добавления задачи и механизма запуска задачи на выполнение. Содержит метод `execute(Runnable)`, который асинхронно запускает задачу на выполнение.

Интерфейс `ExecutorService` позволяет отслеживать выполнение задач и возвращать их результаты. В дополнение к методу `execute()` интерфейса `Executor` содержит методы:

- `Future<T> submit(Callable<T> task)` – запускает асинхронно на выполнение задачу, возвращает объект `Future` для получения результата.
- `List<Future<T>> invokeAll(Collcection<Callable> tasks)` – запускает на выполнение несколько задач, возвращая список объектов `Future` для получения их результатов после завершения всех задач.

Интерфейс `ScheduledExecutorService`, расширяющий интерфейс `ExecutorService` позволяет с помощью метода `ScheduledFuture schedule()` задавать начало выполнения задач.

Вспомогательный класс `Executors` содержит статические методы, возвращающие наиболее часто используемые реализации интерфейсов `ExecutorService` или `ScheduledExecutorService`. В основном это варианты класса `ThreadPoolExecutor`, представляющие из себя пулы потоков:

- `FixedThreadPool` – пул с фиксированным количеством потоков и общей неограниченной очередью.
- `WorkStealingPool` – пул с изменяемым количеством потоков и несколькими очередями, обычно количество потоков равно количеству доступных процессоров.
- `CachedThreadPool` – пул с кэшированием, по возможности повторно используются уже имеющиеся потоки. При этом потоки уничтожаются, если не использовались в течение минуты, и создаются новые по мере необходимости. Удобен для большого количества коротких задач.
- `ScheduledThreadPool` – пул с возможностью запуска задач с задержкой или периодических задач.
- `SingleThreadExecutor` – однопоточный исполнитель с неограниченной очередью. В случае необходимости создается новый поток. Гарантируется, что в один момент времени будет выполняться одна задача, и что они будут выполнены в порядке поступления.
- `SungleThreadScheduledExecutor` – `SingleThreadExecutor` с возможностью задания задержки выполнения задачи.

Кроме этого, можно использовать механизм fork-join с помощью класса `ForkJoinPool` и класса `ForkJoinTask`. Основным принципом использования механизма fork-join является выполнение задачи потоком самостоятельно, если она достаточно мала, в противном случае задача делится на 2 подзадачи, которые передаются на исполнение другим потокам, которые в свою очередь выполняют аналогичную операцию. Выполнение задачи реализуется обычно одним из потомков класса `ForkJoinTask` – `RecursiveAction` или `RecursiveTask`. Оба этих класса имеют абстрактный метод `compute()`, в котором и реализуется алгоритм. Например, для вычисления чисел Фибоначчи:

```
public class FiboTask extends RecursiveTask<Integer> {
    final int n;
    public FiboTask(int number) { n = number; }
    Integer compute() {
        if (n < 2) { return n; }
```

```
        FiboTask f1 = new FiboTask(n – 1);
        f1.fork(); // выдача подзадачи другому потоку
        FiboTask f2 = new FiboTask(n – 2);
        return f2.compute() + f1.join();
    }
}
```

Для запуска задачи с помощью пула потоков создается объект класса `ForkJoinPool`, у которого вызывается метод `invoke()`, которому в качестве параметра передается задача (подкласс `ForkJoinTask`).

Интерфейс `ComletionService` разделяет создание новых асинхронных задач и получение результатов завершенных задач. Он содержит методы:

- `Future<V> submit(Callable<T>)` – запустить задачу на выполнение и вернуть `Future` для получения результата
- `Future<V> poll()` – получить результат очередной задачи или `null`, если нет завершенных задач.
- `Future<V> take()` – получить результат очередной задачи, ожидая завершения, если таких нет.

Очереди (Queues)

В пакете `java.util.concurrent` определены реализации очередей:

- `ConcurrentLinkedQueue` – потокобезопасная реализация очереди, основанной на связанном списке.
- `ConcurrentLinkedDeque` – потокобезопасная реализация двусторонней очереди, основанной на связанном списке.
- `BlockingQueue` – интерфейс, представляющий блокирующую очередь, которая, в дополнение к стандартным методам (`add()`, `remove()`, `element()`, выбрасывающим исключение при невозможности выполнить действие; `offer()`, `poll()`, `peek()`, возвращающим `false` или `null` при невозможности выполнить действие), предоставляет еще два вида методов:
    - `put(e)` и `take()`, которые при невозможности добавить или получить элемент блокируют очередь, пока не удастся выполнить действие;
    - `offer(e, time, unit)` и `poll(time, unit)`, которые при невозможности добавить или получить элемент блокируют очередь на определенное время.
- `LinkedBlockedQueue` и `LinkedBlockingDeque` – реализация блокирующей очереди (одно- и двухсторонней) на основе связанного списка.
- `ArrayBlockingQueue` – реализация блокирующей очереди на основе кольцевого буфера.
- `SynchronousQueue` – реализация блокирующей очереди с синхронизацией добавления и получения элементов. После добавления элемента очередь блокируется, пока элемент не будет получен.

- `PriorityBlockingQueue` – реализация блокирующей очереди с приоритетами, задающимися компаратором.
- `DelayQueue` – реализация блокирующей очереди с задержкой получения элементов после их добавления.
- интерфейс `TransferQueue` и его реализация `LinkedTransferQueue` – расширяет интерфейс `BlockingQueue` путем добавления метода `transfer()`, который блокирует поток, добавивший элемент, до тех пор, пока элемент не будет получен другим потоком.

Потокобезопасные коллекции (Concurrent Collections)

Кроме очередей, пакет содержит набор потокобезопасных коллекций. В отличие от синхронизированных коллекций, которые можно получить с помощью класса `Collections` и методов `getSynchronizedList` и подобных, и которые блокируют доступ ко всей коллекции при вызове методов, потокобезопасные коллекции блокируют коллекцию частично, тем самым увеличивая производительность при параллельных операциях.

- `ConcurrentHashMap` – разрешает любое количество параллельных операций чтения и ограниченное количество параллельных операций записи (достигается изменением распределения значений в хеш-таблице).
- `ConcurrentSkipListMap` – реализация ассоциативного массива на основе списков с пропусками.
- `ConcurrentSkipListSet` – реализация множества на основе списков с пропусками.
- `CopyOnWriteArrayList` – реализация динамического массива, при которой любая модифицирующая операция выполняется с использованием копирования массива.
- `CopyOnWriteArraySet` – реализация множества на основе `CopyOnWriteArrayList`.

Синхронизаторы (Synchronizers)

Набор классов, обеспечивающих различные варианты синхронизации между потоками:

- `Semaphore` – семафор позволяет ограничить количество потоков, имеющих доступ к ресурсу. При создании семафора указывается количество разрешений. Метод `acquire` уменьшает количество разрешений на 1, если они еще есть, и возвращается немедленно. Если разрешений нет, поток блокируется, пока они не появятся. Метод `tryAcquire()` вместо блокировки возвращает `false`, если разрешения отсутствуют. Метод `release()` возвращает в семафор одно разрешение, позволяя «проснуться» одному из потоков, ожидающих его.
- `Exchanger<V>` – класс для обмена данными двух потоков. Метод `V exchange(V)` ожидает, когда второй поток вызовет такой же метод, после чего отдает свое значение, получая взамен значение от второго потока.

- `CountDownLatch` – триггер с обратным отсчетом. При создании объекта данного класса указывается начальное значение счетчика. Вызов метода `await()` приводит к блокировке потока до тех пор, пока необходимое количество раз не будет вызван метод `countDown()`. После этого триггер срабатывает и ожидающие потоки просыпаются.
- `CyclicBarrier` – циклический барьер. При создании барьера указывается его размер. Вызов метода `await()` приводит к блокировке потока до тех пор, пока количество потоков, ожидающих дальнейшего продвижения, не сравняется с размером барьера. После этого потоки разблокируются.
- `Phaser` – синхронизатор, объединяющий возможности триггера с обратным отсчетом и циклического барьера. При его создании можно задать количество потоков, необходимых для преодоления барьера. Потоки могут вызывать методы:
  - `register()`, увеличивающий барьер на 1;
  - `arriveAndAwaitAdvance()` – прибыть к барьеру и ждать остальных;
  - `arriveAndDeregister()` – прибыть к барьеру и отменить дальнейшее участие;
  - `arrive()` – прибыть к барьеру и не ждать его открытия.

  С помощью этих методов можно реализовывать различные сценарии работы.

Блокировки (Locks)

Данные классы и интерфейсы находятся в пакете `java.util.concurrent.locks`. Основными являются:

- Интерфейс `Lock` предназначен для реализации поведения, подобного синхронизированным методам, но с расширенными возможностями, включая неблокирующий захват блокировки, блокировку с прерыванием и блокировку с таймаутом. Методы:
  - `lock()` — получить блокировку. Если блокировка свободна, то она захватывается текущим потоком. Если блокировка захвачена другим потоком, то текущий поток блокируется и «засыпает» до освобождения блокировки.
  - `unlock()` – освободить блокировку.
  - `lockInterruptibly() throws InterruptedException` – получить блокировку с возможностью отменить захват блокировки прерыванием потока.
  - `tryLock()` – получить блокировку, если она свободна.
  - `tryLock(time, unit)` – получить блокировку, если она свободна, в противном случае поток блокируется и спит определенное время, либо до получения блокировки, либо до прерывания.
  - `Condition newCondition()` – возвращает условие, связанное с данной блокировкой

- Интерфейс `Condition` позволяет осуществлять блокировку с ожиданием условия, подобно методам wait-notify, но опять же с расширенными возможностями, например, с возможностью иметь несколько условий для одной блокировки. Методы:
    - `await()` – заставляет текущий поток ожидать сигнала или прерывания.
    - `await(time, unit)` – заставляет текущий поток ожидать сигнала, прерывания, либо окончания таймаута.
    - `awaitUntil(Date)` – заставляет текущий поток ожидать сигнала, прерывания, либо наступления определенного момента времени.
    - `awaitUninterruptibly()` – заставляет текущий поток ожидать сигнала.
    - `signal()` – пробуждает один поток.
    - `signalAll()` – пробуждает все потоки.
- Интерфейс `ReadWtiteLock` – содержит пару связанных блокировок, одну для операций чтения (позволяет получить блокировку, если нет захваченных блокировок записи), другую для операций записи (позволяет получить блокировку только если нет захваченных блокировок как чтения, так и записи). Методы:
    - `readLock()` — возвращает блокировку для чтения.
    - `writeLock()` — возвращает блокировку для записи.
- Класс `ReentrantLock` – реализация интерфейса `Lock` с возможностью повторного получения уже захваченной блокировки (при этом поддерживается счетчик захватов). Для освобождения блокировки нужно освободить ее столько же раз, сколько она была захвачена.
- Класс `ReentrantReadWriteLock` – реализация интерфейса `ReadWriteLock` со следующими характеристиками:
    - По умолчанию включается «нечестный» режим без гарантии порядка предоставления блокировок.
    - Опционально доступен «честный» режим с меньшей производительностью, но с предоставлением блокировок наиболее долго ждущим потокам.
    - Блокировка на чтение предоставляется только в случае отсутствия потока, ожидающего блокировки на запись.
    - Поток, имеющий блокировку на запись, может получить также блокировку на чтение.
    - Поток, имеющий блокировку на чтение, не может получить блокировку на запись.
    - Блокировка на запись поддерживает условия (`Condition`), блокировка на чтение — не поддерживает.
    - Оба вида блокировки являются прерываемыми.

Атомарные объекты (Atomic objects)

Хотя использование переменных с модификатором `volatile` позволяет решить проблему атомарности чтения-записи, другие операции (инкремент, декремент, сложение) остаются неатомарными. Для упрощения работы в таких случаях существует пакет `java.util.concurrent.atomic`. Классы, входящие в данный пакет, позволяют достичь более высокой производительности, чем использование синхронизированных блоков для обеспечения атомарности операций над различными типами данных.

- `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, `AtomicReference` – классы, реализующие атомарный доступ и операции для соответствующих типов:
  - `V get()` – получить значение;
  - `set(V)` – установить значение;
  - `V getAndSet(V)` – установить значение и вернуть старое;
  - `boolean compareAndSet(V expect, V update)` – если значение равно `expect`, установить новое значение в `update`.
  - для `AtomicInteger` и `AtomicLong` дополнительно:
    - `incrementAndGet(), decrementAndGet(), addAndGet()`;
    - `getAndIncrement(), getAndDecrement(), getAndAdd()`.
- `AtomicIntegerArray`, `AtomicLongArray`, `AtomicReferenceArray`:
  - Те же методы, только с индексом элемента.
- `AtomicMarkedReference`, `AtomicStampedReference`:
  - Представляют собой ссылки с дополнительным полем типа `boolean` или `int`.

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,`	Читает не более len байтов в буфер, заполняя его

int offset, int len)	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 возвращает количество прочитанных символов.
`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 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 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. Значащими являются 16 младших битов, старшие — игнорируются. При ошибке ввода-вывода генерируется исключение. Подклассы должны обеспечить 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	`FileInputStream`	`FileOutputStream`	`FileReader`	`FireWriter`
массив	`ByteArray InputStream`	`ByteArray OutputStream`	`CharArray Reader`	`CharArray Writer`
строка	-	-	`StringReader`	`StringWriter`
конвейер	`PipedInputStream`	`PipedOutputStream`	`PipedReader`	`PipedWriter`

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 использованием 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 стандартные 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 сериализации `ObjectStreamConstants.PROTOCOL_VERSION_1` (использовалась в JDK версии 1.1) или `ObjectStreamConstants.PROTOCOL_VERSION_2` (по умолчанию начиная с версии JDK 1.2).
`void defaultWriteObject()`	Вызывается из метода `writeObject` сериализуемого класса для сохранения нестатических и нетранзитивых полей этого класса.
`writeBoolean, writeByte, writeShort, writeChar, writeInt, writeLong, writeFloat, writeDOuble, writeUTF`	Методы, аналогичные методам класса `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 класса 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)`, предна-

значенный для записи и чтения состояния объекта. По умолчанию никакие 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`.

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`	Возвращает `InetAddress` с заданным IP-адресом и

`host, byte[] addr)`	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 миллисекундах. 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 Socket

`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 устанавливаться соединение.	`Socket soc = new Socket(ia, 8888);`
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();`

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 заданного массива байтов с заданной длиной.
`DatagramPacket(byte[] buf, int length, InetAddress addr, int port)`	Создает датаграмму из заданного массива байтов с заданной длиной, для отправления на заданный адрес по заданному 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 использованием первого доступного локального порта.
`DatagramSocket(int port)`	Создает сокет с использованием 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 `Socket-`

	`TimeoutException`.
`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	Создание 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	Преобразование URI в URL.	`URL url = uri.toURL();`

3.1	Открытие URL-соединения и потока данных.	`URLConnection uc = url.openConnection();`
		`uc.connect();`
		`InputStream is = uc.getInputStream();`

3.2	Открытие потока данных.	`InputStream is = url.openStream();`
4	Получение данных.	
5	Закрытие потока и соединения.	`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.

Расширенный ввод-вывод

Чтение и запись данных с помощью потоков ввода-вывода подразумевает побайтовый (посимвольный) обмен данными. При этом данные передаются последовательно, нет возможности произвольно перемещаться по потоку вперед или назад. Чтение и запись производятся синхронно, то есть любой вызов метода `read` или `write` дожидается завершения операции чтения или записи байта. Для преодоления этих ограничений, начиная с версии 1.4, был введен пакет `java.nio`, который позволяет производить обмен данными целыми блоками, производить асинхронный обмен данными, что значительно ускорило операции ввода-вывода при работе с большими объемами данных.

Работа с буферами

Абстрактный класс `java.nio.Buffer` представляет собой контейнер для хранения фиксированного количества элементов данных.
Буфер имеет следующие характеристики:

- Вместимость (`capacity`) — количество элементов в буфере (задается при создании)
- Граница (`limit`) — неотрицательный индекс первого недоступного элемента
- Позиция (`position`) — неотрицательный индекс текущего элемента
- Метка (`mark`) — отмеченная позиция для операции сброса

Характеристики связаны соотношением: `0 ≤ mark ≤ position ≤ limit ≤ capacity`
Значения характеристик можно получить с помощью методов `capacity()`, `limit()` и `position()`. Метод `remaining()` возвращает количество элементов между текущей позицией и границей.
Методы класса `Buffer`, изменяющие значения характеристик буфера, приведены в таблице:
Если ячейка таблицы пустая или условие не выполняется, соответствующая характеристика не изменяется. Символ Ø обозначает, что характеристика сбрасывается и ее значение после выполнения метода не определено.

Метод	Новое значение характеристики		
	`mark`	`position`	`limit`
`limit(newLim)`	Ø, если > `newLim`	`newLim`, если > `newLim`	`newLim`
`position(newPos)`	Ø, если > `newPos`	`newPos`	
`mark()`	`position`		
`reset()`		`mark`	
`clear()`	Ø	`0`	`capacity`

flip()	Ø	0	position
rewind()	Ø	0	
compact()	Ø	remaining()	capacity

Буфер связан с массивом, где физически хранятся данные. Получить ссылку на этот массив можно с помощью метода `array()`. Каждый потомок класса `Buffer` определяет 2 статических метода для создания буфера: `allocate(int capacity)` и `wrap(array)`. Метод `allocate` в качестве параметра получает вместимость буфера, создает массив заданного размера и возвращает ссылку на буфер. Метод `wrap` в качестве параметра получает ссылку на массив, который будет использован для хранения данных, связывает этот массив с буфером и возвращает ссылку на буфер.

Методы для чтения данных называются `get()`, методы для записи — `put()`.

Операции обмена данными с буфером могут быть с абсолютной и относительной индексацией. Операции с относительной индексацией работают с текущей позицией, то есть записывают данные в текущую позицию или читают данные из текущей позиции. Позиция при этом изменяется. Относительные операции делятся на одиночные и групповые. Одиночные операции читают и записывают один элемент данных, групповые — читают или записывают массив элементов.

Операции с абсолютной индексацией читают и записывают элемент данных по указанному индексу, не изменяя значение текущей позиции. Они читают или записывают только один элемент.

Простейший алгоритм работы с буфером:

1. Создать буфер с помощью метода `allocate`, указав его вместимость.
2. Вызвать метод `clear()`, чтобы очистить буфер перед записью данных. При этом значение границы устанавливается равным вместимости — буфер полностью доступен для записи.
3. С помощью одного из методов `put()` заполнить буфер данными. При этом значение текущей позиции будет равно количеству записанных в буфер элементов.
4. Вызвать метод `flip()`, чтобы переключить буфер в режим чтения. При этом значение границы станет равным значению позиции, то есть прочитать можно будет ровно столько элементов, сколько было записано. Позиция переместится на 0, чтобы читать элементы с начала буфера.
5. С помощью одного из методов `get()` прочитать данные. Методом `hasRemaining()` можно проверять, есть ли еще доступные для чтения элементы.
6. Если необходимо еще раз прочитать содержимое буфера, можно вызвать метод `rewind()`, который сбросит позицию в начало буфера, но оставит неизменной границу.
7. Если были прочитаны не все данные, но надо записать еще, то буфер можно сжать с помощью метода `compact()`. Он переносит оставшуюся часть недочитанных данных между позицией и границей в начало буфера,

устанавливают новую позицию в конец этих данных (то есть позиция будет равна значению `remaining()`), а границу передвигают на конец буфера.
8. В случае необходимости можно повторить пункты, начиная с 2.

Для буферов также определены методы:
`duplicate()` - дублирует буфер, создавая новый объект для доступа к тем же данным. Все изменения в одном буфере будут видны в буфере-дубле. Значения вместимости, границы, позиции и метки нового буфера независимы от соответствующих значений исходного, но при создании дубля они будут идентичны значениям исходного.
`slice()` - дублирует часть буфера, создавая новый буфер для доступа к данным, начиная с текущей позиции исходного буфера. Вместимость нового буфера будет равна количеству оставшегося места в исходном буфере (от позиции до границы). Значения вместимости, границы, позиции и метки нового буфера независимы от соответствующих значений исходного

Для каждого типа данных определен свой класс буфера. Класс `ByteBuffer` представляет байтовый буфер, классы `CharBuffer`, `ShortBuffer`, `IntBuffer`, `LongBuffer`, `FloatBuffer`, `DoubleBuffer` — соответственно, буферы для других типов данных. `ByteBuffer` применяется намного чаще, так как основной единицей обмена данными является байт. Поэтому данный класс содержит дополнительные методы:

1. Для чтения данных других типов — `getChar()`, `putChar()`, `getShort()`, `putShort()`, и так далее. Эти методы могут быть относительными и абсолютными.
2. Для представления байтового буфера как буфера другого типа — `asCharBuffer()`, `asIntBuffer()`, `asDoubleBuffer()` и так далее.

Кроме того, байтовый буфер можно создать с прямым доступом (с помощью метода `allocateDirect()`). В этом случае виртуальная машина постарается создать буфер так, чтобы операционная система имела к нему прямой доступ на чтение и запись. При этом буфер можно располагаться вне основной памяти Java-машины. Будет затрачено больше ресурсов на создание и удаление буфера, но при этом скорость операций ввода-вывода существенно возрастет.

Кодировки символов

Для представления символов Java использует стандарт Unicode. Однако, часто бывает, что работать приходится с текстами в других кодировках, многие из которых требуют один байт для представления одного символа. Для представления кодировки служит класс `java.nio.charset.Charset`. В нем определены методы:

- `CharBuffer decode(ByteBuffer b)` – преобразует текст в заданной кодировке в Unicode;
- `ByteBuffer encode(CharBuffer b)` – преобразует текст из Unicode в заданную кодировку.

Каналы

Канал представляет из себя объект, способный эффективно передавать данные между буфером и устройством ввода-вывода (файлом или сокетом). Существуют 2 типа каналов — файловые и сетевые.
В пакете `java.nio.channels` определен интерфейс `Channel`, а также классы для различных типов каналов/

Файловые каналы

Объект типа `FileChannel` можно получить, вызвав метод `open()` класса `FileChannel`, либо метод `getChannel()` у одного из открытых объектов типа `FileInputStream`, `FileOutputStream` или `RandomAccessFile`.
Для чтения из канала в буфер предназначен метод `read(ByteBuffer)`. Для записи из буфера в канал — метод `write(ByteBuffer)`. Методы возвращают количество прочитанных или записанных байт. Значение «`-1`» обозначает конец файла.
Есть специальный вид буфера с прямым доступом, который позволяет еще больше ускорить ввод-вывод — буфер, отображенный в память (`MappedByteBuffer`). Его можно получить с помощью метода `map()` класса `FileChannel`. У такого буфера определены 3 метода:

- `load()` — загружает содержимое буфера в память;
- `isLoaded()` — проверяет находится ли содержимое буфера в памяти (без гарантии истинности результата);
- `force()` — заставляет сохранить в файле все изменения, которые были сделаны в буфере.

Сетевые каналы

Для обмена данными по протоколу TCP предназначен канал `SocketChannel`. Он может быть получен двумя способами:

1. На стороне сервера сначала создается объект класса `ServerSocketChannel` с помощью метода `open()` класса `ServerSocketChannel`, затем он связывается с портом с помощью метода bind(), которому передается объект класса `InetSocketAddress`, содержащий номер порта для прослушивания. Далее, метод `accept()` возвращает объект класса `SocketChannel`, как только будет получен запрос от клиента.
2. На стороне клиента вызывается метод `open()` класса `SocketChannel`, затем метод `connect()`, которому передается объект класса `InetSocketAddress`, содержащий адрес узла сети и номер порта.

Полученный объект класса `SocketChannel` используется для обмена данными через буфер — с помощью методов `read(ByteBuffer)` и `write(ByteBuffer)`.

Для обмена данными по протоколу UDP предназначен канал `DatagramChannel`. Его можно использовать следующим образом: вызывается метод open() класса

DatagramChannel, затем он связывается с локальным портом с помощью метода bind(), которому передается объект класса InetSocketAddress, содержащий номер порта.

Далее метод send(ByteBuffer, InetSocketAddress) используется для отправки данных по адресу и номеру порта, заданным в объекте InetSocketAddress. Метод receive(ByteBuffer) предназначен для приема данных.

Если адрес и номер порта при обмене не меняются, то, чтобы не указывать их каждый раз при отправке, можно создать условное «соединение» с помощью метода connect(InetSocketAddress). Реального соединения при этом не создается, а вместо методов send и receive становится возможными использовать методы read(ByteBuffer) и write(ButeBuffer).

RMI – вызов удаленных методов

RMI (Remote method invocation) – технология распределенного объектного взаимодействия, позволяющая объекту, расположенному 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

Структура RMI приведена на рис. 8

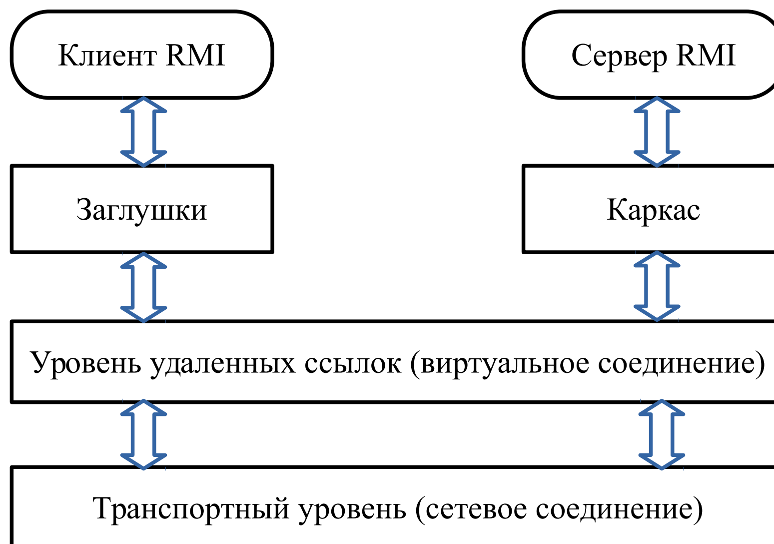


Рисунок 8. Структура RMI

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 быть `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.

```
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
```

Интернационализация и локализация

Интернационализация

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

Характеристики интернационализированного приложения:

1. один и тот же код может работать в любой местности при условии добавления данных о локализации;
2. приложение отображает текст на родном языке конечного пользователя;
3. текстовые элементы не являются частью кода, а хранятся отдельно и запрашиваются динамически;
4. поддержка новых языков не требует перекомпиляции;
5. данные, зависящие от местности, такие как даты и денежные единицы, отображаются в соответствии с регионом и языком конечного пользователя;
6. 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:

1. Текст.
2. Числа.
3. Денежные единицы.
4. Дата и время.
5. Изображения.
6. Цвета.
7. 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. Строки хранятся в кодировке Latin-1, для преобразования можно использовать утилиту `native2ascii`, входящую в набор JDK.

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. 9.

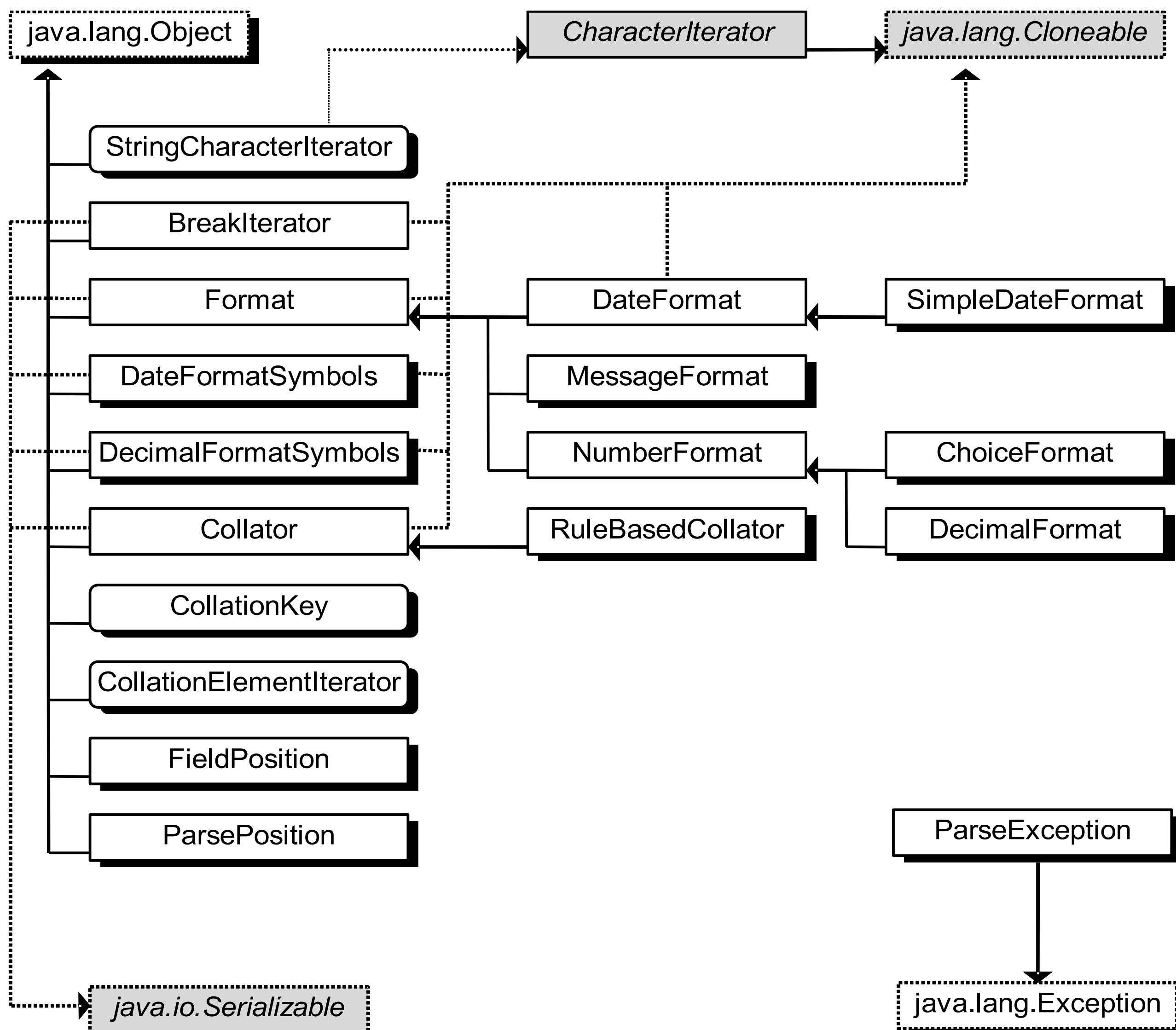


Рисунок 9. Пакет `java.text`.

Класс NumberFormat

Абстрактный класс, позволяющий форматировать числа, денежные единицы, проценты в соответствии с форматом, принятым в определенной местности. Список допустимых местностей, для которых определены форматы, можно получить с помощью статического метода `NumberFormat.getAvailableLocales()`.

Форматирование осуществляется в 2 этапа:

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. Список допустимых местностей, для которых определены форматы, можно получить с помощью статического метода `DateFormat.getAvailableLocales()`.
Форматирование осуществляется в 2 этапа:

1. Получение требуемого экземпляра класса с помощью одного из методов
    - `getDateInstance`;
    - `getTimeInstance`;
    - `getDateTimeInstance`.
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 (2nd 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 **<u>1:15 PM</u>** on **<u>April 13, 1998</u>**, we detected **<u>7</u>** spaceships on the planet **<u>Mars</u>**.
2. Помещение шаблона сообщения в ResourceBundle:

```
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());
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() + "  ");
}
```

Рефлексия

Рефлексия позволяет получить информацию о полях, методах и конструкторах загруженных классов и созданных объектов, а также модифицировать эту информацию в определенных пределах. Эти функции необходимы для создания компиляторов, интерпретаторов, отладчиков. Также рефлексия используется при сериализации объектов для получения иерархии ссылок.

Рефлексия реализуется с помощью класса `java.lang.Class` и классов и интерфейсов из пакета `java.lang.reflect`.

Класс Class

Виртуальная машина хранит ссылку на один объект класса `Class` для каждого загруженного и используемого в программе класса. Получить ссылку на объект класса `Class` можно одним из следующих способов:

1. У любого объекта нужного класса вызвать метод `getClass()`.
2. Вызвать статический метод `Class.forName(String name)`.
3. Воспользоваться конструкцией `.class`, которая возвращает объект класса `Class` для любого ссылочного или примитивного типа;
4. Для примитивных типов в соответствующих оболочках есть константа `TYPE`.

Пример:

```
Class c1 = "This is a string".getClass();
Class c2 = Class.forName("java.lang.System");
Class c3 = Double.class;
Class c4 = Integer.TYPE; // то же самое, что int.class
```

Методы класса Class:

- `getName()` – возвращает имя класса. Для массивов имя состоит из одной или нескольких открывающих квадратных скобок (количество определяется размерностью массива), латинской заглавной буквы, определяющей тип элемента массива (`B` – `byte`, `C` – `char`, `S` – `short`, `I` – `int`, `J` – `long`, `F` – `float`, `D` – `double`, `Z` – `boolean`, `L - ссылка`), и, если элементы имеют ссылочный тип — наименование этого типа. Например, [[`Z` – обозначение имени класса для двумерного массива с элементами `boolean`, `[Ljava.lang.Object` – обозначение имени класса для одномерного массива элементов типа `Object`.
- `isPrimitive()`, `isArray()`, `isEnum()`, `isInterface()`, `isAnnotation()` – возвращают true, если класс представляет соответственно примитивный тип, массив, перечисление, интерфейс, аннотацию.
- `isMemberClass()`, `isLocalClass()`, `isAnonymousClass()` – возвращают `true`, если класс является соответственно вложенным, локальным, анонимным.
- `getPackage()` – возвращает пакет, к которому принадлежит данный класс.

- getSuperclass() – возвращает родительский класс данного класса. Для интерфейсов, примитивных типов, или класса, представляющего Object, возвращает null. Для массивов возвращает класс, представляющий Object.
- getInterfaces() – для класса возвращает массив интерфейсов, которые реализует данный класс, для интерфейса — массив интерфейсов-предков данного интерфейса. Если таковых не существует, то массив имеет нулевую длину.
- getFields(), getMethods(), getConstructors(), getClasses() — возвращают соответственно список полей, методов, конструкторов или вложенных классов и интерфейсов, которые определены в данном классе и его предках. Данные методы возвращают только общедоступные элементы, имеющие модфикатор public. Для получения списка всех полей, методов, конструкторов, вложенных классов и интерфейсов, включая элементы с модификаторами protected, private и с пакетным доступом, используются методы getDeclaredFields(), getDeclaredMethods(), getDeclaredConstructors(), getDeclaredClasses().
- isInstance(Object obj) — проверяет, может ли заданный объект быть приведен к типу, представленному данным классом.
- cast(Object obj) — приводит объект к типу, представленному данным классом.
- newInstance() — создает объект данного класса путем вызова конструктора без параметров.
- getEnumConstants() - в случае, если класс представляет из себя перечисление, данный метод возвращает массив возможных значений перечислимого типа.

Интерфейс Member

Интерфейс java.lang.reflect.Member реализуется классами Field, Method и Constructor. В нем объявлены следующие методы:

- getName() — возвращает имя элемента;
- getDeclaringClass() — возвращает класс, где объявлен данный элемент;
- getModifiers() — возвращает модификаторы в виде целого числа.

Класс Modifier

Для представления модификаторов классов, интерфейсов, полей, методов и конструкторов служит класс java.lang.reflect.Modifier. В нем определен набор статических методов и констант для проверки и кодирования модификаторов. Метод getModifier(), определенный для классов Class, Field, Method, Constructor возвращает целое число, представляющее собой битовую маску всех модификаторов данного элемента. Значения, соответствующие модификаторам, приведены в таблице:

бит	11	10	9	8	7	6	5	4	3	2	1	0
целое значение	2048	1024	512	256	128	64	32	16	8	4	2	1
модификатор	strictfp	abstract	interface	native	transient	volatile	synchronized	final	static	protected	private	public

Методы класса `Modifier`:

- 12 методов `isPublic(int mod)`, `isPrivate(int mod)`, и так далее — возвращают `true`, если в `mod` установлен соответствующий модификатор.
- Методы `fieldModifiers()`, `methodModifiers()`, constructorModifiers(), `interfaceModifiers()`, `classModifiers()` — возвращают все возможные модификаторы для полей, методов, констукторов, интефейсов и классов соответственно.
- Метод `toString()` — возвращает список модификаторов в виде строки.

Класс Field

Класс `java.lang.reflect.Field` обеспечивает доступ к полю, а также позволяет получать и изменять значение поля.

В классе `Field`, кроме уже описанных методов интерфейса `Member`, дополнительно определены следующие методы:

- `getType()` — возвращает тип поля.
- `Object get(Object obj)` — возвращает значение поля у объекта `obj`. Если поле — статическое, то ссылка на объект игнорируется. Если поле имеет примитивный тип, оно упаковывается в объект. Также в классе `Field` определен набор методов, возвращающих значение определенного примитивного типа — `getByte()`, `getChar()`, `getInt()` и т. д. Работают они аналогично.
- `set(Object obj, Object value)` — устанавливает полю объекта `obj` новое значение `value`. Если поле статические, то ссылка на объект игнорируется. Если поле имеет примитивный тип, его значение распаковывается. Также в классе `Field` определен набор методов, устанавливающих значение определенного примитивного типа — `setByte()`, `setChar()`, `setInt()` и так далее. Работают они аналогично.

Класс Method

Класс `java.lang.reflect.Method` обеспечивает доступ к методу, позволяет получать значения типа результата, параметров и исключений, а также вызывать метод.

В классе Method, кроме уже описанных методов интерфейса Member, дополнительно определены следующие методы:

- getReturnType() — возвращает тип результата метода.
- getParameterTypes() — возвращает массив типов параметров метода.
- getExceptionTypes() — возвращает массив исключений, которые может выбрасывать метод.
- Object invoke(Object obj, Object... args) — вызывает метод у указанного объекта obj с аргументами args и возвращает полученный результат в виде Object. Если метод статический, то obj игнорируется. Если какие-то аргументы или результат имеют примитивный тип, то они подвергаются упаковке или распаковке.

Класс Constructor

Класс java.lang.reflect.Consturctor<T> обеспечивает доступ к конструктору, позволяет получать значения типа параметров и исключений, а также вызывать конструктор для создания нового экземпляра класса.

В классе Constructor, кроме уже описанных методов интерфейса Member, дополнительно определены следующие методы:

- getParameterTypes() и getExceptionTypes() аналогичны уже рассмотренным методам класса Method.
- T newInstance(Object... args) — вызывает соответствующий конструктор с аргументами args и возвращает полученный результат параметризованного типа. Если какие-то аргументы имеют примитивный тип, то они подвергаются упаковке.

Класс Array

Класс java.lang.reflect.Array определяет набор статических методов для динамического создания и доступа к массивам.

Метод isArray() класса Class позволяет удостовериться, что данный класс представляет собой массив. В классе Array определены следующие статические методы:

- Object newInstance(Class type, int length) — создает массив элементов типа type размером length и возвращает ссылку на созданный массив.
- Object newInstance(Class type, int... dimensions) — аналогичен предыдущему, но предназначен для массивов любой размерности. Длина каждой размерности задается отдельно.
- int getLength(Object array) — возвращает длину массива.
- Семейство методов get и set, аналогичных методам класса Field, дополнительно имеющие аргумент index типа int для получения или установки значения элементов массива.

УНИВЕРСИТЕТ ИТМО

Миссия университета – генерация передовых знаний, внедрение инновационных разработок и подготовка элитных кадров, способных действовать в условиях быстро меняющегося мира и обеспечивать опережающее развитие науки, технологий и других областей для содействия решению актуальных задач.

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
Харитонова Анастасия Евгеньевна
Цопа Евгений Алексеевич

Программирование на языке Java

Конспект лекций

Издание 2-е, исправленное и дополненное

В авторской редакции
Редакционно-издательский отдел Университета ИТМО
Зав. РИО Н.Ф. Гусарова
Подписано к печати
Заказ №
Тираж
Отпечатано на ризографе

Редакционно-издательский отдел
Университета ИТМО
197101, Санкт-Петербург, Кронверкский пр., 49