Программирование на Java

Лекция 13. Пакет java.lang.

20 апреля 2003 года

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>

Аннотация

В этой лекции рассматривается основная библиотека Java – java.lang. В ней содержатся классы Object и Class, классы-обертки для примитивных типов, класс Math, классы для работы со строками String и StringBuffer, системные классы System, Runtime и другие. В этом же пакете находятся типы, уже рассматриваемые ранее – для работы с исключительными ситуациями и потоками исполнения.

Оглавление

Лекция 13. Пакет java.lang.

Содержание лекции.

1. Введение

В состав пакета java.lang входят классы, составляющие основу для всех других - из-за чего он является наиболее важным из всех, входящих в Java API. Каждый класс в Java неявным образом (по умолчанию) импортирует все классы этого пакета - можно не писать import java.lang.*;.

Основу пакета составляют классы:

Object - является корневым в иерархии классов. Если при описании класса не указывается родительский, то им считается класс Object. Все объекты, включая массивы, наследуются от этого класса.

Class - экземпляры этого класса представляют классы и интерфейсы в запущенной Java-программе.

String - представляет средства работы с символьными строками

StringBuffer - используется для работы (создания) строк

Number - абстрактный класс, являющийся суперклассом для классов-объектных оберток числовых примитивных типов Java

Character - объектная обертка для типа char

Boolean- объектная обертка для типа boolean

Math - реализует ряд наиболее известных математических функций

Throwable - базовый класс для объектов, представляющих исключения. Любое исключение, которое может быть брошено и, соответственно, перехвачено блоком catch, должно быть унаследовано от Throwable

Thread - представляет вычислительный поток в Java. Виртуальная машина Java позволяет одновременно выполнять несколько потоков

ThreadGroup - позволяет объединять потоки в группу, и производить действия сразу над всеми потоками в ней. Сущствуют ограничения по безопасности на манипуляции с потоками из других групп. операции над потоком могут проводиться только потоком, относящимся к той же группе.

System - содержит полезные поля и методы для работы системного уровня

Runtime - позволяет приложению взаимодействовать с окружением в котором оно запущено

Process - представляет внешнюю программу, запущенную при помощи Runtime

ClassLoader - отвечает за загрузку классов

SecurityManager - для обеспечения безопасности накладывает ограничения на данную среду выполнения программ

Compiler - используется для поддержки Just-in-Time компиляторов

Интерфейсы:

Cloneable - должен быть реализован объектами, которые планируется клонировать с помощью средств JVM (Object.clone())

Runnable - может использоваться в сочетании с классом Thread, позволяя реализовать метод run(), котрый будет вызван при старте данного потока.

Comparable - позволяет упорядочивать (сортировать, сравнивать) объекты каждого класса, реализующего этот интерфейс.

2. Object

Класс Object является базовым для всех остальных классов. Он определяет методы, которые поддерживаются любым классом в Java.

Это методы:

1.0) public boolean equals(Object obj) - определяет, являются ли объекты одинаковыми. Если оператор == определяет равенство именно объектных ссылок, то метод equals() определяет, содержат ли объекты одинаковую информацию. Реализация этого метода в классе Object такова, что значение true будет возвращено тогда и только тогда, если объектные ссылки переданного объекта и объекта, у которого вызывается метод совпадают, то есть:

```
public boolean equals(Object obj) {
  return (this == obj);
 }
```

В классах - наследниках, этот метод при необходимости может быть переопределен, что бы отражать действительное равенство объектов. Например, сравнение объектов-оберток целых чисел (класс Integer), должно по всей логике возвращать значение true, если равны значения int чисел, которые обернуты, даже если различаются объектные ссылки на сами объекты класса-обертки Integer.

Метод equals() может быть переопределен любым способом (например, всегда возвращает false, или, наоборот, - true) - компилятор, конечно же, не будет проводить анализ реализации и давать рекомендации, НО: реализация метода предполагается с учетом следующих правил:

1. рефлексивность: для любой объектной ссылки x, вызов x.equals(x) возвращает true
2. симметричность: для любых объектных ссылок x и y, вызов x.equals(y) возвращает true тогда и только тогда, если вызов y.equals(x) возвращает true
3. транзитивность: для любых объектных ссылок x, y и z, если x.equals(y) возвращает true и y.equals(z) возвращает true, тогда вызов x.equals(z) должен вернуть true
4. непротиворечивость: для любых объектных ссылок x и y, многократные последовательные вызовы x.equals(y) возвращают одно и то же значение (либо всегда true либо всегда false)
5. для любой не равной null объектной ссылки x, вызов x.equals(null) должен вернуть значение false

Пример:

```
package demo.lang;
```

```
public class Rectangle {
 public int sideA;
 public int sideB;
public Rectangle(int x, int y) {
 super();
 sideA = x;
 sideB = y;
}
public boolean equals(Object obj) {
 if(!(obj instanceof Rectangle)) return false;
 Rectangle ref = (Rectangle)obj;
 return (((this.sideA==ref.sideA)&&(this.sideB==ref.sideB))||
    (this.sideA==ref.sideB)&&(this.sideB==ref.sideA));
}
public static void main(String[] args) {
 Rectangle r1 = new Rectangle(10,20);
 Rectangle r2 = new Rectangle(10,10);
 Rectangle r3 = new Rectangle(20,10);
 System.out.println("r1.equals(r1) == " + r1.equals(r1));
 System.out.println("r1.equals(r2) == " + r1.equals(r2));
 System.out.println("r1.equals(r3) == " + r1.equals(r3));
 System.out.println("r2.equals(r3) == " + r2.equals(r3));
 System.out.println("r1.equals(null) == " + r1.equals(null));
}
}
```

Запуск этой программы, очевидно, приведет к выводу на экран следующего:

```
r1.equals(r1) == true
r1.equals(r2) == false
r1.equals(r3) == true
r2.equals(r3) == false
r1.equals(null) == false
```

В этом примере метод equals() у класса Retangle был переопределен таким образом, что бы прямоугольники были равны, если их можно наложить друг на друга (геометрическое равенство).

1.1) public int hashCode() - возвращает хеш-код (hash code) для объекта. Хеш-код - это целое число, которое с очень большой вероятностью является уникальным для данного объекта. Это значение используется классом Hashtable для хранения с возможностью быстрой выборки объектов(хеш-код используется для группировки объектов, и при поиске объекта достаточно просмотреть только блок, соответствующий его хеш-коду)

Обычно вполне можно пользоваться стандартной реализацией, но если же будет принято решение изменить метод вычисления хеш-кода, то необходимо убедиться, что в этом случае будет возвращаться одно и то же значение для равных между собой объектов. То есть, если x.equals(y) возвращает true, то хеш-коды x и y должны совпадать, то есть вызовы

x.hashCode() и y.hashCode() должны возвращать одно и то же значение. В противном случае Hashtable будет считать объекты различными, не вызывая метод equals()

Формально правила, которым должна следовать реализация метода, формулируется следующим образом:

a) в одном запуске программы, для одного объекта, при вызове метода hashCode(), должно возвращаться одно и то же int значение, если между этими вызовами НЕ были затронуты данные, используемые для проверки объектов на идентичность в методе equals(). Это число НЕ обязано быть одним и тем же при повторном запуске той же программы, даже если все данные будут идентичны

b) если два объекта идентичны, то есть вызов метода equals(Object) возвращает true, тогда вызов метода hashCode() у каждого из этих двух объектов должен возвращать одно и то же значение

c) если два объекта различны, то есть вызов метода equals(Object) возвращает false, тогда различие их хеш-кодов желательно, но НЕ обязательно. Различие в хеш-кодах для НЕ идентичных объектов нужно только для обеспечения хорошей производительности при использовании этих объектов в хеш-таблицах.

В классе Object реализация метода hashCode() использует для получения результата преобразование внутреннего адреса объекта в памяти, поэтому если не перекрывать эту реализацию, то для разных объектов будут возвращены различные значения.

1.2) public String toString() - возвращает строковое представление объекта. В классе Object этот метод реализован следующим образом:

```
public String toString() {
  return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

То есть возвращает строку, содержащую название класса объекта и его хеш-код.

В классах-наследниках этот метод может быть переопределен для получения более наглядного пользовательского представления объекта. Обычно это значения некоторых полей, характеризующих экземпляр. Например, для книги это может быть название, автор и количество страниц:

```
package demo.lang;

public class Book {
 private String title;
 private String author;
 private int pagesNumber;
public Book(String title, String author, int pagesNumber) {
 super();
 this.title = title;
 this.author = author;
 this.pagesNumber = pagesNumber;
}
public static void main(String[] args) {
```

```
 Book book = new Book("Java2", "Sun", 1000);
 System.out.println("object is: " + book);
}
public String toString(){
 return "Book: " + title + " ( " + author + ", " + pagesNumber + " pages )";
}
}
```

При запуске этой программы, на экран будет выведено следующее:

```
object is: Book: Java2 ( Sun, 1000 pages )
```

То есть, для получения строкового представления объекта book, был вызван его метод toString() (оператор "+" определен для строк таким образом, что все операнды будут сначала приведены к String, для объектов в этом случае вызывается метод toString()).

1.3) wait(), notify(), notifyAll() - эти методы используются для поддержки многопоточности. Они определены с атрибутом final и НЕ могут быть переопределены в классах-наследниках.

1.4) protected void finalize() throws Throwable - этот метод вызывается Java-машиной перед тем, как garbage collection (сборщик мусора) очистит память, занимаемую объектом. Работа сборщика мусора и описание схемы, по которой производится вызов метода finaliuze() приведено в соответствующей главе. Здесь же ограничимся только напоминанием, что объект считается пригодным для сборщика мусора, когда выполняющейся части программы не будет доступно ни одной ссылки на объект. Перед очисткой занимаемой объектом памяти, будет произведен вызов его метода finalize().

Реализация этого метода в классе Object - не производит никаких действий. В классах-наследниках этот метод может быть переопределен для проведения всех необходимых действий по освобождению различных занимаемых ресурсов - закрытия сетевых соединений, файлов и т.д.

1.5) protected native Object clone() throws CloneNotSupportedException - создает копию объекта. Для того, что бы им можно было воспользоваться, объект должен реализовывать интерфейс Cloneable. Этот интерфейс не определяет никаких методов, определение, что класс его реализует - только символизирует, что можно создавать копии объектов этого класса. В классе Object метод clone() реализован таким образом, что будут скопированы только базовые типы и ссылки на объекты. Если же потребуется "глубокое " копирование, то есть скопировать не только ссылки на объекты, но и создать копии объектов - в классах-наследниках метод clone() можно переопределить.

Например, в следующем примере:

```
package demo.lang;
public class BookStorage implements Cloneable{
 public Book[] books;
public BookStorage() {
 books = new Book[2];
 books[0] = new Book("Essential java", "Sun", 500);
 books[1] = new Book("Professional Java", "Sun", 1000);
}
}
```

Если выполнить следующий код:

```
 BookStorage storage1 = new BookStorage();
 try{
  BookStorage storage2 = (BookStorage)storage1.clone();
  storage2.books[2] = new Book("Java 2 Enterprise", "Sun", 2000);
System.out.println("storage1.books[1] = " + storage1.books[1]);
 }catch(CloneNotSupportedException e){
  e.printStackTrace();
 }
```

То изменения затронут и объект storage1 - изменится содержимое массива books. Это произойдет, потому что реализация метода clone() по умолчанию скопирует только ссылку на массив, поэтому изменение его содержимого затронет оба объекта: storage1 и storage2. Если же мы хотим, что бы копировались не только ссылки, имеющиеся в объекте, но объекты, на которые имеются ссылки, необходимо соответствующим образом переопределить метод clone(). В случае данного примера, это можно сделать следующим образом:

```
protected Object clone() throws CloneNotSupportedException{
 BookStorage result = (BookStorage)super.clone();
 result.books = (Book[])books.clone();
 return result;
}
```

Теперь будет создаваться копия не только BookStorage, но и массива books. Это легко проверить, еще раз запустив вышеприведенный код.

1.6) public final native Class getClass() - возвращает объект типа Class, соответствующий классу объекта. Именно этот объект используется при использовании синхронизации статических методов.

3. Class

В запущенной программе Java каждому классу соответствует объект типа Class. Этот объект содержит информацию, необходимую для описания класса – поля, методы и т.д.

Класс Class не имеет открытого конструктора – объекты этого класса создаются автоматически Java-машиной по мере загрузки классов и вызовов метода defineClass() загрузчика классов. Получить экземпляр Class для конкретного класса можно воспользовавшись методом forName():

2.1) public static Class forName(String name, boolean initialize, ClassLoader loader) – возвращает объект Class, соответствующий классу или интерфейсу с названием, указанным в name (необходимо указывать полное название класса или интерфейса), используя переданный загрузчик классов. Если в качестве загрузчика классов loader передано значение null, будет взят таковой, который использовался для загрузки вызывающего класса. При этом класс будет инициализирован, только если значение initialize равно true и класс не был инициализирован ранее.

Довольно часто проще и удобнее воспользоваться методом forName(), передав только название класса:

public static Class forName(String className) – при этом будет использоваться загрузчик вызывающего класса, и класс будет инициализирован, если до этого не был.

2.2) public Object newInstance() – создает и возвращает объект класса, который представляется данным экземпляром Class. Создание будет проходить, используя конструктор без параметров. Если такового в классе нет, будет брошено исключение InstantiationException. Это же исключение будет брошено, если объект Class соответствует абстрактному классу, интерфейсу или же по какой-либо другой причине.

2.3) Каждому методу, полю, конструктору класса так же соответствуют объекты, которые можно получить вызовом соответствующих методов на объекте Class:

getMethods(), getFields(), getConstructors(),getDeclaredMethods() и т.д. В результате будут получены объекты, которые отвечают за поля, методы, конструкторы объекта. Их можно использовать для формирования динамических вызовов Java – такой процесс называется отражение (reflection). Классы, используемые для отражения содержатся в пакете java.lang.reflection.

Динамическое создание экземпляров не обязательно всегда сопровождается вызовом методов посредством отражения. Это показано в следующем примере:

```
package demo.lang;
interface Vehicle {
void go();
}
class Automobile implements Vehicle {
public void go() {
  System.out.println("Automobile go!");
}
}
class Truck implements Vehicle {
public Truck(int i) {
  super();
}
public void go() {
  System.out.println("Truck go!");
}
}
public class VehicleStarter {
public static void main(String[] args) {
 Vehicle vehicle;
 String[] vehicleNames = {"demo.lang.Automobile", "demo.lang.Truck",
"demo.lang.Tank"};
 for(int i=0; i<vehicleNames.length; i++){
  try{
   String name = vehicleNames[i];
   System.out.println("look for clas for: " + name);
   Class aClass = Class.forName(name);
```

```
   System.out.println("creating vehicle...");
   vehicle = (Vehicle)aClass.newInstance();
   System.out.println("create vehicle: " + vehicle.getClass());
   vehicle.go();
  }catch(ClassNotFoundException e){
   System.out.println("Exception: " + e.toString());
  }catch(InstantiationException e){
   System.out.println("Exception: " + e.toString());
  }catch(Throwable th){
   System.out.println("Another problem: " + th.toString());
  }
 }
}
}
```

Если запустить эту программу, на экран будет выведено следующее:

```
look for clas for: demo.lang.Automobile
creating vehicle...
create vehicle: class demo.lang.Automobile
Automobile go!
look for clas for: demo.lang.Truck
creating vehicle...
Instantiation exception: java.lang.InstantiationException
look for clas for: demo.lang.Tank
Class not found: java.lang.ClassNotFoundException: demo.lang.Tank
```

Как видим, объект класса Automobile был успешно создан, а дальше с ним работа велась через интерфейс Vehicle. А вот класс Truck был найден, но при создании объекта этого класса, было брошено и, соответственно, обработано исключение java.lang.InstantiationException. Так-как класс java.lang.Tank не был определен, то при попытке получить объект Class, ему соответствующий, было брошено исключение java.lang.ClassNotFoundException.

4. Wrapper Classes

Во многих случаях бывает предпочтительней работать именно с объектами, а не примитивными типами. Так, например, при использовании коллекций, просто необходимо значения примитивных типов каким-то образом представлять в виде объектов. Для этих целей и предназначены так называемые классы-обертки. Для каждого примитивного типа Java существует свой класс обертка. Такой класс является неизменяемым (то есть, для изменения значения необходимо создавать новый объект), к тому же имеет атрибут final - от него нельзя наследовать класс. Все классы-обертки (кроме Void) реализуют интерфейс Serializable, поэтому объекты любого (кроме Void) класса-обертки могут быть сериализованы. Все классы-обертки содержат статическое поле TYPE - содержащее объект Class, соответствующий примитивному оборачиваемому типу.

Так же классы-обертки содержат статические методы для обеспечения удобного манипулирования соответствующими примитивными типами, например преобразование к строковому виду.

В таблице 1 описаны примитивные типы и соответствующие им классы обертки:

Класс-обертка	Примитивный тип
Byte	byte
Short	short
Character	char
Integer	int
Long	long
Float	float
Double	double
Boolean	boolean

При этом классы обертки числовых типов - Byte, Short, Integer, Long, Float, Double наследуются от одного класса - Number. В нем содержится код, общий (часть реализована посредством абстрактных методов) для всех классов-оберток числовых типов - примитивного значения в виде byte, short, int, long, float и double.

Все классы-обертки реализуют интерфейс Comparable. Number реализует интерфейс java.io.Serializable, поэтому все объекты классов-оберток примитивных числовых типов могут быть сериализованы.

Все классы-обертки числовых типов имеют метод equals(Object), сравнивающий примитивные значения объектов.

> Стоит быть особо внимательным - результат выполнения (new Integer(1)).equals(new Byte(1)) дает false, хотя сами значения, вокруг которых обернуты объекты - равны. Такой результат получается потому, что во всех таких классах, метод equals() определен таким образом, что сначала производится проверка, совпадают ли типы (классы) значений, и если нет - сразу возвращается false. Например в jdk1.3.1 для Integer метод equals() определен следующим образом:

```
    public boolean equals(java.lang.Object obj) {
    if(obj instanceof java.lang.Integer)
        return value == ((java.lang.Integer)obj).intValue();
    else
        return false;
}
```

Рассмотрим более подробно некоторые из классов-оберток.

4.1. Integer

Наиболее часто используемые статические методы.

public static int parseInt(String s) - преобразует в int значение строку, представляющую десятичную запись целого числа

public static int parseInt(String s, int radix) - преобразует в int значение строку, представляющую запись целого числа в системе счисления radix

Оба этих метода могут возбуждать исключение NumberFormatException, если строка, переданная на вход, содержит нецифровые символы.

Не следует путать эти методы, с другой парой похожих методов:

public static Integer valueOf(String s) public static Integer valueOf(String s,int radix)

Эти методы выполняют аналогичную работу, только результат представляют в виде объекта-обертки.

Существует так же два конструктора для создания экземпляров в класса Integer

Integer(String s) - такой метод, принимающий в качестве параметра строку, представляющую значение, имеется для каждого класса

Integer(int i) - аналогично, для каждого класса-обертки числового примитивного типа, имеется конструктор, принимающий значение оборачиваемого примитивного типа

Первый вариант конструктора так же может возбуждать исключение NumberFormatException

pubic static String toString(int i) - используется для преобразования значения типа int в строку

Далее перечислены методы, преобразующие int в строковое восьмеричное, двоичное и шестнадцатеричное представление:

pubic static String toOctalString(int i) - восьмеричное

pubic static String toBinaryString(int i) - двоичное

pubic static String toHexString(int i) - шестнадцатиричное .

Имеется так же две статические константы:

Integer.MIN_VALUE - минимальное int значение

Integer.MAX_VALUE - максимальное int значение.

Аналогичны константы, равные границам соответствующих типов, определены и для всех остальных классов-оберток числовых примитивных типов.

public int intValue();

возвращает значение примитивного типа для данного Integer. Классы-обертки остальных примитивных целочисленных типов: Byte, Short, Long содержат только аналогичные методы и константы, только определенные для соответствующих типов: byte, short, long.

Пример:

```
    public static void main(String[] args) {
        int i = 1;
        byte b = 1;
        String value = "1000";
        Integer iObj = new Integer(i);
        Byte bObj = new Byte(b);
        System.out.println("while i==b is " + (i==b));
        System.out.println("iObj.equals(bObj) is " + iObj.equals(bObj));
        Long lObj = new Long(value);
        System.out.println("lObj = " + lObj.toString());
        Long sum = new Long(lObj.longValue() + iObj.byteValue() +
bObj.shortValue());
        System.out.println("The sum = " + sum.doubleValue());
    }
```

В данном примере произвольным образом используются различные варианты классов-оберток и их методов. Соответственно, в результате выполнения, на экран будет выведено следующее:

```
while i==b is true
iObj.equals(bObj) is false
lObj = 1000
The sum = 1002.0
```

Оставшиеся классы-обертки числовых типов: Float и Double - помимо описанного для целочисленных примитивных типов, дополнительно содержат определения следующих констант:

NEGATIVE_INFINITY - отрицательная бесконечность

POSITIVE_INFINITY - положительная бесконечность

NaN - НЕ числовое значение (неопределенность, комплексное число и т.д.)

Кроме того, несколько иначе трактуется значение MIN_VALUE - вместо наименьшего значения, оно представляет минимальное положительное (строго > 0) значение, которое может быть представлено соответствующим примитивным типом и, соответственно, классом-оберткой.

Кроме классов-оберток для примитивных числовых типов, определены таковые и для остальных примитивных типов Java:

4.2. Character

Реализует интерфейс Comparable.

Из конструкторов - имеет только один, принимающий char в качестве параметра.

Кроме стандартных методов equals(), hashCode(), toString() еще из НЕ статических, содержит только два метода:

public char charValue() - возвращает обернутое значение char

public int compareTo(Character anotherCharacter) - сравнивает обернутые значения char как числа, то есть возвращает значение return this.value - anotherCharacter.value;

Так-же, для совместимости с интерфейсом Comparable, метод compareTo() определен с параметром Object:

public int compareTo(Object o) - если переданный объект имеет тип Character, результат будет аналогичен вызову compareTo((Character)o), иначе будет брошено исключение ClassCastException - так-как Character можно сравнивать только с Character.

Статических методов в классе Character довольно много, все их здесь перечислять смысла не имеет. Все они могут быть довольно полезны. Большинство - это методы, принимающие char и проверяющие всевозможные свойства - является ли цифрой, буквой, буквой заглавного или строчного шрифта, может ли с него начинаться переменная в Java, и т.д. Например:

public static boolean isDigit(char c) - проверяет, является ли char цифрой

public static boolean isLetter(char c) - проверяет, является ли char буквой

public static boolean isDigitOrLetter(char c) - проверяет, является ли char цифрой или буквой

public static boolean isIdentifierStart(char c) - проверяет, является ли символ подходящим для того, что бы с него начиналось наименование переменной JAVA

Эти методы возвращают значение истина или ложь, в соответствии с тем выполнен ли критерий проверки.

4.3. Boolean

Представляет класс-обертку для примитивного типа boolean.

Реализует интерфейс java.io.Serializable и во всем напоминает аналогичные классы-обертки.

Для получения примитивного типа используется метод booleanValue().

4.4. Void

Этот класс-обертка, в отличии от остальных, НЕ реализует интерфейс java.io.Serializable. Он не имеет открытого конструктора. Более того, экземпляр этого класса вообще не может быть получен. Он нужен только для получения ссылки на объект Class, соответствующий void. Эта ссылка представлена статической константой TYPE.

Делая краткое заключение по классам-оберткам, можно сказать что

- каждый примитивный тип имеет соответствующий класс-обертку
- все классы-обертки могут быть сконструированы как с использованием примитивных типов, так и с использованием String, за исключением Character, который может быть сконструирован только по char
- Классы-обертки могут сравниваться с использованием метода equals()
- примитивные типы могут быть извлечены из классов-оберток с помощью соответствующего метода xxxxValue() (например intValue())
- классы-обертки так же являются классами-утилитами, т.е. предоставляют набор статических методов для работы с примитивными типами
- классы-обертки не могут быть модифицированы

5. Math

Класс Math состоит из набора статических методов, производящих наиболее популярные математические вычисления и двух констант, имеющих особое значение в математике - это число Пи и экспонента. Часто этот класс еще называют классом-утилитой (Utility class). Так как все методы класса статические нет необходимости создавать экземпляр этого класса - поэтому он и не имеет открытого конструктора. Нельзя так же и унаследовать этот класс, поскольку он объявлен с атрибутом final.

Итак, константы определены следующим образом:

public static final double Matht.PI - задает число Пи

public static final double Matht.E - число e.

> Следует обратить внимание, что тип констант double. При использовании этих констант в вычислениях, результат будет автоматически конвертирован в double, если его явно не привести к другому типу.

В таблице 2 приведены все методы класса. Так же дано их краткое описание

static double	abs(double a)	Возвращает абсолютное значение типа double
static float	abs(float a)	Возвращает абсолютное значение типа byte
static int	abs(int a)	Возвращает абсолютное значение типа int (1)
static long	abs(long a)	Возвращает абсолютное значение типа long
static double	acos(double a)	Вернет значение арккосинуса угла в диапазоне от 0 до PI
static double	asin(double a)	Вернет значение арксинуса угла в диапазоне от -PI/2 до PI/2
static double	atan(double a)	Вернет значение арктангенса угла в диапазоне от -PI/2 до PI/2
static double	ceil(double a)	Возвращает наименьшее целое число которое больше a. (2)
static double	floor(double a)	Возвращает целое число которое меньше a. (2)
static double	cos(double a)	Возвращает косинус угла (3)
static double	IEEEremainder(double a, double b)	Возвращает остаток от деления a/b по стандарту IEEE 754 (* см. пояснение дальше по тексту)
static double	sin(double a)	Возвращает косинус угла (3)
static double	tan(double a)	Возвращает тангенс угла (3)
static double	exp(double a)	Возвращает e в степени числа a
static double	log(double a)	Возвращает натуральный логарифм числа a
static double	max(double a, double b)	Возвращает наибольшее из двух чисел типа double (4)
static float	max(float a, float b)	Возвращает наибольшее из двух чисел типа double (4)
static long	max(long a, long b)	Возвращает наибольшее из двух чисел типа long (4)
static int	max(int a, int b)	Возвращает наибольшее из двух чисел типа int (4)
static double	min(double a, double b)	Возвращает наименьшее из двух чисел типа double (4)
static float	min(float a, float b)	Возвращает наименьшее из двух чисел типа double (4)
static long	min(long a, long b)	Возвращает наименьшее из двух чисел типа long (4)
static int	min(int a, int b)	Возвращает наименьшее из двух чисел типа int (4)
static double	pow(double a, double b)	Возвращает a в степени b
static double	random()	Возвращает случайное число в диапазоне от 0.0 до 1.0
static double	rint(double a)	Возвращает int число, ближайшее к a

static long	round(double a)	Возвращает значение типа long ближайшее по значению к a. (5)
static long	round(double a)	Возвращает значение типа long ближайшее по значению к a. (6)
static double	sqrt(double a)	Возвращает положительный квадратный корень числа a
static double	toDegrees(double angrad)	Преобразует значение угла из радианов в градусы
static double	toRadians(double angdeg)	Преобразует значение угла из градусов в радианы

Следует обратить внимание, что

1. abs вернет значения типа int, если в качестве параметра будут переданы значения типа byte, short, char.
2. Угол задается в радианах.
3. Если round имеет аргумент double, то возвращается значение типа long, если аргумент типа float, то будет возвращено значение типа int.
4. Операторы и функции для вычисления остатка подробно рассматривались в 3 лекции.

6. Строки

6.1. String

Этот класс используется в Java для представления строк. Он предназначен для хранения не модифицируемых строк. После того как создан экземпляр этого класса, строка уже не может быть модифицирована. Для создания объекта String можно использовать различные варианты конструктора. Наиболее простой - если содержимое строки известно на этапе компиляции - это написать текст в кавычках:

```
String abc = "abc";
```

Можно использовать и различные варианты конструктора. Наиболее простой из них - конструктор, получающий на входе строковый литерал.

```
String s = new String("immutable");
```

На первый взгляд, эти варианты создания строк отличаются только синтаксисом. На самом деле различие есть, хотя в большинстве случаев его можно и не заметить. Каждый строковый литерал имеет внутреннее представление, как экземпляр класса String. Классы в JAVA могут иметь целый набор таких строк и, когда класс компилируется, экземпляр представляющий литерал, добавляется в этот набор. Однако, если такой литерал уже имеется где-то в другом месте класса, т.е. уже представлен в наборе строковых объектов, то новый экземпляр (фактически копирующий уже существующий) создан не будет. Вместо этого будет создана ссылка на уже имеющийся объект. Т.к. строки не являются модифицируемыми объектами, то это не нанесет никакого урона другим фрагментам программы. С другой стороны, если объект-строка создается с помощью явного вызова

конструктора, то даже если эти строки будут совершенно идентичными, экземпляры класса String будут отличаться.

В объекте String определен метод equals() который сравнивает две строки на предмет идентичности.

Рассмотрим пример

```
public class Test {
    public Test() {
    }
    public static void main(String[] args) {
        Test t = new Test();
        String s1 = "Hello world !!!";
        String s2 = "Hello world !!!";
        System.out.println("String`s equally = " + (s1.equals(s2)));
        System.out.println("Strings are the same = " + (s1==s2));
    }
}
```

в результате на консоль будет выведено

```
String`s equally = true
Strings are the same = true
```

Теперь несколько модифицируем код

```
public class Test {
    public Test() {
    }
    public static void main(String[] args) {
        Test t = new Test();
        String s1 = "Hello world !!!";
        String s2 = new String("Hello world !!!");
        System.out.println("String`s equally = " + (s1.equals(s2)));
        System.out.println("Strings are the same = " + (s1==s2));
    }
}
```

в результате на консоль будет выведено

```
String`s equally = true
Strings are the same = false
```

В первом примере для создания строк используются строковые литералы, поэтому ссылки s1 и s2 ссылаются на один и тот же объект. Во втором случае применены конструкторы, поэтому, несмотря на то, что строки идентичны переменные ссылаются на разные объекты, которые, в сущности, создаются во время выполнения программы и не находятся во множестве строковых констант, которое создается на момент компиляции.

Следует обратить внимание, что при создании экземпляров строк во время выполнения, они не помещаются в набор строковых констант. Однако, можно явно указать на необходимость поместить, вновь создаваемый экземпляр класса String в этот набор, применив метод intern()

```
public class Test {
    public static void main(String[] args) {
        Test t = new Test();
        String s1 = "Hello world!!!";
        String s2 = new String("Hello world!!!").intern();
        System.out.println("String`s equally = " + (s1.equals(s2)));
        System.out.println("Strings are the same = " + (s1==s2));
    }
}
```

в этом случае на консоль будет выведено

```
String`s equally = true
Strings are the same = true
```

В такой подход экономит занимаемую программой память, что в некоторых случаях может быть существенно. Помимо этого, если мы уверены, что все троки находятся в наборе сформированном при компиляции, для сравнения сток, вместо метода equals() можно использовать оператор ==, который выполняется значительно быстрее.

В JAVA для строк переопределен оператор +. При использовании этого оператора производится конкатенация строк. В классе String так же определен метод

```
public String concat(String s);
```

он возвращает новый объект строку дополненный справа строкой s.

Следует еще раз обратить внимание, что строки являются не модифицируемыми объектами. И если используется оператор конкатенации или какой-либо вспомогательный метод класса String то изменения строки не произойдет, а будет создан новый экземпляр класса String. Так же следует обратить на использование в методах параметров типа String. Не смотря на то, что String является объектом и передается в метод по ссылке, String не модифицируемый объект и все изменения в методе не повлекут изменений исходного объекта.

```
public class Test {
    public static void main(String[] args) {
        Test t = new Test();
        String s = "prefix";
        System.out.println("String before = " + s);
        t.perform(s);
        System.out.println("String after =" + s);
    }
    private void perform(String s){
        System.out.println(s + " suffix");
    }
}
```

```
String before = prefix
prefix suffix
String after =prefix
```

В этом примере видно, что строка s в действительности не изменяется.

Рассмотрим другой пример.

```
public class Test {
    public static void main(String[] args) {
        Test t = new Test();
        String s = " prefix !";
        System.out.println(s);
        s = s.trim();
        System.out.println(s);
        s = s.concat(" suffix");
        System.out.println(s);
    }
}
```

```
prefix !
prefix !
prefix ! suffix
```

В данном случае может сложиться впечатление, что строку можно изменять. В действительности это не так. После выполнения операций trim и concat создается новый объект - строка, ссылка s, будет указывать на новый объект-строку.

Как уже отмечалось ранее, строка состоит из шестнадцатибитовых UNICODE символов. Однако во многих случаях требуется работать с восьмибитовыми символами (ввод/вывод, работа с базой данных и т.д.). Преобразование строки в последовательность байтов (восьмибитовые символы) производится с методами

byte[] getBytes(); - возвращает последовательность байтов, в кодировке принятой по умолчанию. (Как правило зависит от настроек операционной системы)

byte[] getBytes(String encoding); - возвращает последовательность байтов, в кодировке encoding;

Для выполнения обратной операции (преобразования байтов в строку) необходимо сконструировать новый объект-строку.

String(byte[] bytes); - создает строку из последовательности байтов в кодировке принятой по умолчанию;

String(byte[] bytes,String enc); - создает строку из последовательности байтов в указанной кодировке.

6.2. StringBuffer

Этот класс используется для создания и модификации строковых выражений, которые после можно превратить в String. Он реализован на основе массива char[] и, в отличии от String, после создания объекта, значение строки, в нем содержащейся может быть изменено.

Рассмотрим наиболее часто используемые конструкторы класса StringBuffer

StringBuffer () - создает пустой StringBuffer

StringBuffer(String s) - в качестве параметра используется объект String

StringBuffer (int capacity) - создает экземпляр класса StringBuffer с заранее заданным размером. Задание размера не означает, что нельзя будет манипулировать со строками, длинной более указанной при создании объекта, а всего лишь говорит о том, что при манипулировании строками, длина которых меньше указанной, не потребуется дополнительного выделения памяти.

Типичный пример использования StringBuffer может быть продемонстрирован следующим кодом:

```
public class Test {
    public static void main(String[] args) {
        Test t = new Test();
        String s = new String("ssssss");
        StringBuffer sb = new StringBuffer("bbbbbb");
        s.concat("-aaa");
        sb.append("-aaa");
        System.out.println(s);
        System.out.println(sb);
    }
}
```

В результате на экран будет выведено следующее:

```
ssssss
bbbbbb-aaa
```

В данном примере можно заметить, что объект String остался неизменным, а объект StringBuffer изменился.

Основные методы, используемые для модификации StringBuffer - это:

public StringBuffer append(String str) - добавляет переданную строку str к уже имеющейся в объекте;

public StringBuffer insert(int offset, String str) - вставка строки, начиная с позиции offset (пропустив offset символов)

Стоит обратить внимание, что оба этих метода имеют варианты, принимающие в качестве параметров различные примитивные типы Java вместо String. При использовании этих методов, значение этого примитивного типа сначала будет представлено строкой, как это произошло бы вызовом метода String.valueOf(), после чего будет вызван этот же метод, передав полученное значение.

Еще один важный момент, связанный с этими методами - они возвращают сам объект, на котором вызываются. Таким образом, возможно их использование в цепочке. Например:

В результате на экран будет выведено:

```
abcdef
```

При передаче экземпляра класса в качестве параметра в метод StringBuffer, так же следует помнить об отличии String и StringBuffer.

```
public class Test {
    public static void main(String[] args) {
        Test t = new Test();
        StringBuffer sb = new StringBuffer("aaa");
        System.out.println("Before = " + sb);
        t.doTest(sb);
        System.out.println("After = " + sb);
    }

    void doTest(StringBuffer theSb){
        theSb.append("-bbb");
    }
}
```

В результате на экран будет выведено следующее:

```
Before = aaa
After = aaa-bbb
```

Т.к. все объекты передаются по ссылке, в методе doTest, при выполнении операций с theSB, будет модифицирован объект, на который ссылается sb. Следует еще раз напомнить, что для String переопределен оператор +, т.е. если + применить экземплярам класса String то будет осуществлена конкатенация строк и, если один из операндов не принадлежит к классу String, то он будет неявно приведен к этому типу. Примитивные типы будут преобразованы в String, как это произошло бы вызовом метода String.valueOf().

Например

```
System.out.println("1" + 5)  выведет на консоль 15
System.out.println(1+ 5)  выведет на консоль 6
```

7. Системные классы

Следующие классы, которые стоит рассмотреть, отвечают за выполнение хода программы, это

ClassLoader - загрузчик классов. Содержит методы, необходимые для динамической загрузки новых классов

SecurityManager - менеджер безопасности. Содержит различные методы проверки, допустима ли запрашиваемая операция.

System - содержит набор статических методов, применимых к среде, в которой выполняется приложение. Многие из них присутствуют так же в классе Runtime.

Runtime - позволяет приложению взаимодействовать с окружением в котором оно запущено. Каждому приложению соответствует один экземпляр Runtime.

Process - представляет внешнюю программу, запущенную при помощи Runtime

7.1. ClassLoader

Это абстрактный класс, ответственный за загрузку классов. По имени класса он находит либо генерирует данные, которые составляют определение класса. Обычно для этого используется следующая стратегия: название класса преобразуется в название файла - "class file", из которого и считывается вся необходимая информация.

Каждый объект Class содержит ссылку на объект ClassLoader, посредством которого он был загружен.

Для изменения способа загрузки классов, можно реализовать свой загрузчик классов, унаследовав его от ClassLoader. Так, хотя обычно классы загружаются из файлов, однако бывают и другие ситуации. Например, классы могут загружаться через сетевое соединение. Метод defineClass() преобразует массив байт в экземпляр класса Class. Экземпляры полученного таким образом класса могут быть получены, используя метод newInstance() у объекта Class. Методы объектов, полученных с помощью загрузчика классов, могут ссылаться на другие, доступные в запущенном приложении классы. Для получения классов, на которые можно ссылаться, вызывается метод loadClass у загрузчика классов.

Для иллюстрации использования загрузчика классов, приведем пример, как может выглядеть простая реализация загрузчика классов, использующего сетевое соединение:

```
class NetworkClassLoader extends ClassLoader {
    String host;
    int port;
    public NetworkClassLoader(String host, int port) {
        this.host = host;
        this.port = port;
    }
    public Class findClass(String className) {
        byte[] bytes = loadClassData(className);
        return defineClass(className, bytes, 0, bytes.length)
    }
    private byte[] loadClassData(String className) {
       byte[] result = null;
       // open connection, load the class data
       return result;
    }
}
```

В этом примере только показано, что подкласс загрузчика классов доложен определить и реализовать методы findClass() и loadClassData() для загрузки классов. Когда набор байт, образующих класс, загружен, необходимо использовать метод defineClass() для создания класса. Для простоты, в примере приведен только шаблонный код без реализации получения байт из сетевого соединения.

Для получения экземпляров классов, загруженных с помощью этого загрузчика, можно написать код, аналогичный следующему:

```
try{
  ClassLoader loader = new NetworkClassLoader(host, port);
  Object main = loader.loadClass("Main").newInstance();
 }catch(ClassNotFoundException e){
  e.printStackTrace();
 }catch(InstantiationException e){
  e.printStackTrace();
 }catch(IllegalAccessException e){
  e.printStackTrace();
 }
```

Если такой класс не будет найден - будет брошено исключение ClassNotFoundException, если класс будет найден, но произойдет какая-либо ошибка при создании объекта этого класса - будет брошено исключение InstantiationException, и, наконец, если у вызывающего потока не имеется достаточно прав для создания экземпляров этого класса (что будет проверено менеджером безопасности), будет брошено исключение IllegalAccessException.

7.2. SecurityManager - менеджер безопасности

С помощью этого класса приложения могут перед выполнением потенциально опасных операций, определить, является ли операция таковой и может ли она быть выполнена в данном контексте.

Класс SecurityManager содержит много методов с именами, начинающимися с приставки check . Эти методы вызываются различными из библиотек Java перед тем как в них будут выполнены потенциально опасные операции. Типичный такой вызов выглядит примерно следующим образом:

```
SecurityManager security = System.getSecurityManager();
if(security != null){
   security.checkX(…);
}
```

Где X - какой-либо запрос на доступ: Access, Read, Write, Connect, Delete, Exec, Listen и так далее.

Предотвращение вызова производится путем бросания исключения - SecurityException, если вызов операции НЕ разрешен (кроме метода checkTopLevelWindow, который возвращает boolean значение).

Для установки менеджера безопасности в качестве текущего, вызывается метод setSecurityManager() в классе System. Соответственно, для его получения, нужно вызвать метод getSecurityManager().

В большинстве случаев, если приложение запускается локально - будут разрешены все действия. В основном менеджер безопасности проявляет себя при работе с апплетами - загруженными из сети.

7.3. System

Содержит набор полезных статических методов и полей. Экземпляр этого класса НЕ может быть получен. Среди прочих полезных средств, предоставляемых этим классом, особо стоит отметить потоки стандартных ввода и вывода, поток для вывода ошибок; доступ к внешне определенным свойствам; возможность загрузки файлов и библиотек; утилиту для быстрого копирования порций массивов.

Конечно, наиболее широко используемым является стандартный вывод, доступный через переменную System.out. стандартный вывод можно перенаправить в другой поток (файл, массив байт и т.д., главное, что бы это был объект PrintStream):

```
    public static void main(String[] args) {
     System.out.println("Study Java");
     try{
      PrintStream print = new PrintStream(new
FileOutputStream("d:\\file2.txt"));
      System.setOut(print);
      System.out.println("Study well");
     }catch(FileNotFoundException e){
      e.printStackTrace();
     }
    }
```

При запуске этого кода, на экран будет выведено только

```
Study Java
```

И в файл "d:\file2.txt" будет записано

```
Study well
```

Абсолютно аналогично могут быть перенаправлены стандартный ввод System.in вызовом System.setIn(InputStream) и System.err - вызовом System.setErr(PrintStream).

Следующие методы класса System позволяют работать с некоторыми параметрами системы:

public static void runFinalizersOnExit(boolean value) - выставляет, будет ли производиться вызов метода finalize() у всех объектов (у кого еще не вызывался), когда выполнение программы будет окончено

public static native long currentTimeMillis() - возвращает текущее время. Это время представляется как количество миллисекунд, прошедших с 1-го января 1970 года

public static String getProperty(String key) - возвращает значение свойства с названием key. Что бы получить все свойства, какие определены в системе, можно воспользоваться методом

public static java.util.Properties getProperties() - возвращает объект java.util.Properties, в котором содержатся значения всех определенных системных свойств.

7.4. Runtime

Каждому приложению Java сопоставляется экземпляр класса Runtime. Этот объект позволяет взаимодействовать с окружением, в котором запущена Java программа. Получить соответствующий приложению объект Runtime можно вызовом статического метода в этом же классе - Runtime.getRuntime().

Объект этого класса позволяет:

public void exit(int status) - осуществляет завершение программы с кодом завершения status (при использовании этого метода особое внимание нужно уделить обработке исключений - выход будет осуществлен моментально, и в конструкциях try-catch-finally управление в finally передано не будет)

public native void gc() - сигнализирует сборщику мусора о необходимости запуска

public native long freeMemory() - возвращает количество свободной памяти. В некоторых случаях это количество может быть увеличено, если вызвать у объекта Runtime метод gc()

public native long totalMemory() - возвращает суммарное количество памяти, выделенное Java машине. Это количество может из изменяться даже в течении одного запуска, что зависит от реализации платформы на которой запущена Java машина. Так-же, не стоит закладываться на объем памяти, занимаемой одним определенным объектом - эта величина так же зависит от реализации Java машины.

public void loadLibrary(String libname) - загружает библиотеку с указанным именем. Обычно загрузка библиотек производится следующим образом: в классе, использующем native реализации методов, добавляется статический инициализатор, например:

```
static { System.loadLibrary("LibFile"); }
```

Таким образом, когда класс будет загружен и инициализирован, необходимый код для реализации native методов так-же будет загружен. Если будет произведено несколько вызовов загрузки библиотеки с одним и тем-же именем - произведен будет только первый, а все остальные будут проигнорированы.

public void load(String filename) - подгружает файл с указанным названием в качестве библиотеки. В принципе, этот метод работает так-же как и метод load(), только принимает в качестве параметра именно название файла, а не библиотеки, тем самым позволяя загрузить любой файл с native кодом.

public void runFinalization() - производит запуск выполнения методов finalize() у всех объектов, этого ожидающих

public Process exec(String command) - в отдельном процессе запускает команду, представленную переданной строкой. Возвращаемый объект Process может быть использован для управления выполнением этого процесса.

7.5. Process

Объекты этого класса получаются вызовом метода exec() у объекта Runtime - запускающего отдельный процесс. Объект класса Process может использоваться для управления процессом и получения информации о нем.

Process - абстрактный класс, определяющий, какие методы должны присутствовать в реализациях для конкретных платформ. Так, объекты класса Process дают возможность:

public InputStream getInputStream() - получить поток ввода из процесса(это будет Piped поток, присоединенный к потоку стандартного выводы процесса)

getErrorStream(), getOutputStream() - методы, аналогичные getInputStream(), но получающие, соответственно стандартные потоки - ошибок и вывода

public void destroy() - уничтожает процесс. Все подпроцессы, запущенные из него, так же будут уничтожены.

public int exitValue() - возвращает код завершения процесса. По соглашению, код завершения равный 0 - означает нормальное завершение.

public int waitFor() - вынуждает текущий поток выполнения приостановиться до тех пор, пока не будет завершен процесс, представленный этим экземпляром Process. Возвращает значение кода завершения процесса.

Даже если в приложении Java не будет ни одной ссылки на объект Process - процесс не будет уничтожен, и будет продолжать асинхронно выполняться до своего завершения. Спецификацией не оговаривается механизм, с помощью которого будет выделяться процессорное время на выполнение процессов Process и потоков Java. Поэтому при проектировании программ НЕ стоит закладываться ни на какой из их них, так-как реализации могут отличаться в зависимости от реализации Java машины, под которой приложение будет запущено.

8. Потоки исполнения

В Java поддерживает многопоточность. То есть программа может выполняться в нескольких потоках выполнения команд, которым определенным образом выделяется процессорное время. Всевозможное управление ходом выполнения потоков, и организация их взаимодействия производится с помощью классов Runnable, Thread, ThreadGroup.

8.1. Runnable

Runnable - это интерфейс, содержащий один единственный метод без параметров: run(). При использовании объектов, реализующих этот интерфейс, при старте нового потока, управление в нем будет передано именно в этот метод. Любой класс, объекты которого планируется запускать отдельным потоком должны реализовывать этот интерфейс.

8.2. Thread

Объекты этого класса представляют потоки.

Что бы реализовать выполнение некоторого кода в отдельном потоке, можно пойти двумя путями:

а) Написать класс, реализующий интерфейс Runnable, поместив код для выполнения в метод run(). Создать объект класса Thread, передав конструктору объект этого класса. Запустить выполнение в отдельном потоке, вызвав у объекта Thread метод start().

Пример:

```
public static void main(String[] args) {
    class RunCode implements Runnable{
     public void run(){
      System.out.println("RunCode.run() begins");
      System.out.println("RunCode.run() ends");
     }
    }
    Runnable run = new RunCode();
    Thread thread = new Thread(run);
    thread.start();
    System.out.println("main finish");
}
```

В результате выполнения на экран может быть получено примерно следующее:

```
RunCode.run() begins
main finish
RunCode.run() ends
```

Не обязательно именно в таком порядке - фраза "main finish" может быть выведена как самой первой, так и самой последней. Это обусловлено тем, что в метод run() в объекте класса RunCode выполняется в отдельном потоке. Так-как спецификацией механизм распределения процессорного времени между потоками не ограничен жесткими рамками, то может получиться так, сначала выполнится метод run(), и только после этого управление будет передано главному потоку. А может получиться и наоборот - сначала закончит выполнение главный поток и только потом управление будет передано методу run() объекта RunCode.

b) Написать класс, унаследовав его от Thread. При этом в метод run() поместить код, который должен выполняться в отдельном потоке. После этого достаточно будет создать объект этого класса, и вызвать у него метод start().

Предыдущий пример, тогда будет выглядеть следующим образом:

```
public static void main(String[] args) {
    class RunCode extends Thread{
     public void run(){
      System.out.println("RunCode.run() begins");
      System.out.println("RunCode.run() ends");
     }
    }
    Thread thread = new RunCode();
    thread.start();
    System.out.println("main finish");
}
```

Функциональных различий в использовании двух приведенных путях создания новых потоков нет. Класс Thread реализует интерфейс Runnable, и если объект этого класса был создан без передачи конструктору объекта Runnable, в качестве этого объекта будет использоваться сам объект Thread (через ссылку this).

Итак, для управления выполнением потока, класс Thread имеет следующие методы:

public void start() - производит запуск выполнения нового потока

public final void join() - приостанавливает выполнение потока, из которого был вызван этот метод до тех пор, пока не закончит выполнение поток, у объекта Thread которого был вызван этот метод

public static void yield() - поток, из которого вызван этот метод, временно приостановится(отдаст процессорное время), что бы дать возможность выполняться другим потокам

public static void sleep(long millis) - поток, из которого вызван этот метод, перейдет в состояние "сна" на время millis миллисекунд, после чего сможет продолжить выполнение. При этом нужно учесть, что поток именно сможет продолжить выполнение, а НЕ продолжит. То есть через время millis миллисекунд, этому потоку может быть выделено процессорное время (механизм распределения определяется реализацией Java-машины). Правильней было бы говорить, что поток продолжит выполнение НЕ раньше чем через время millis миллисекунд.

еще несколько методов, которые, объявлены deprecated, и рекомендуется избегать их использовать. Это: suspend()-временно прекратить выполнение, resume() - продолжить выполнение (приостановленное вызовом suspend()), stop() - остановить выполнение потока

При вызове метода stop(), в потоке, который представляет этот объект Thread, будет брошена ошибка ThreadDeath. Это ошибка, класс которой ThreadDeath унаследован от Error. Если ошибка не будет обработана в программе и, соответственно, произойдет прекращение работы потока, в top-level обработчике сообщения о ненормальном завершении выведено не будет, так-как такое завершение рассматривается как нормальное. Если в программе эта ошибка будет перехвачена и обработана (например, произведены некоторые действия по закрытию потоков и т.д.), то очень важно позаботиться о том, что бы эта же ошибка была брошена дальше - что бы поток действительно закончил свое выполнение. Класс ThreadDeath специально унаследован от Error, а не Exception, так-как очень часто используется перехват исключений именно класса Exception, и, таким, образом, поток может продолжить выполнение.

Так же Thread позволяет выставлять такие свойства потока, как:

Name - значение типа String, которое можно использовать для более наглядного обращения с потоками в группе

Daemon - выполнение программы не будет прекращено до тех пор пока выполняется хотя бы один НЕ daemon поток

Priority - определяет приоритет потока. В классе Thread определены константы, задающие минимальное и максимальное значения для приоритетов потока - MIN_PRIORITY и MAX_PRIORITY , а так же значение приоритета по умолчанию - NORM_PRIORITY .

Эти свойства могут быть изменены только до того момента, когда поток будет запущен, то есть, вызван метод start() объекта Thread.

Получить эти значения, можно, конечно же, в любой момент жизни потока - и после его запуска, и после прекращения выполнения. Так же, можно и узнать, в каком состоянии сейчас находится поток - вызовом методов isAlive() - выполняется ли еще, isInterrupted() - прерван ли.

8.3. ThreadGroup

Для того, что бы отдельный поток не мог оказаться "невоспитанным" и начать останавливать и прерывать все потоки подряд, введено понятие группы. Поток может оказывать влияние только на потоки, которые находятся в одной с ним группе. Группу потоков представляет класс ThreadGroup. Такая организация позволяет защитить потоки от нежелательного внешнего воздействия. В группе потоков так же могут содержаться другие группы потоков, и так далее, организуя, таким образом, некоторое дерево, в котором каждый объект ThreadGroup, за исключением коневого, имеет родителя.

9. Исключения

Подробно механизм использования исключений описан в соответствующей лекции. Здесь остановимся только на том, что базовым классом для всех исключений является класс Throwable. Любой класс, который планируется использовать как исключение, должен явным или неявным образом быть от него унаследованным. Класс Throwable, а так же наиболее значимые его наследники - классы: Error, Exception, RuntimeException содержатся именно в пакете java.lang.

10. Заключение

В этой главе Вы получили представление о назначении и возможностях классов, представленных в пакете java.lang. Как Вы теперь знаете, пакет java.lang автоматически импортируется во все Java программы и содержит фундаментальные классы и интерфейсы, которые составляют основу для других пакетов Java.

Были рассмотрены все наиболее важные классы пакета java.lang:
Object, Class – основные классы, представляющие объект и класс объектов;

классы-обертки (Wrapper классы) – так как многие классы работают именно с объектами, иногда бывает необходимо представлять значения примитивных типов Java в виде объектов; в таких случаях используют классы-обертки;

Math – класс, предоставляющий набор статических методов, реализующих наиболее распространенные математические функции;

String и StringBuffer – классы для работы со строками;

System, Runtime, Process, ClassLoader, SecurityManager – системные классы, помогающие взаимодействовать с программным окружением (System, Runtime, Process), загружать классы в JVM (ClassLoader) и управлять безопасностью (SecurityManager);

Thread, ThreadGroup, Runnable – типы, обеспечивающие работу с потоками исполнения в Java;

Throwable, Error, Exception, RuntimeException – базовые классы для всех исключений.

11. Контрольные вопросы

13-1. В чем различие между следующими кусками кода:

1.
```
String s1 = "abc";
String s2 = new String("abc");
boolean result = (s1==s2);
```

2.
```
String s1 = new String("abc");
String s2 = new String("abc");
boolean result = (s1.equals(s2));
```

a.) Значение result в первом случае будет равно false, в то время как во втором – true. В первом случае сравниваются значения ссылок s1 и s2, во втором, проверяется, идентичны ли объекты, на которые указывают эти ссылки. При этом s1 и s2 указывают на разные объекты, так как объект s2 был получен применением оператора new, вследствие чего был создан новый объект. В случае если бы s2 был получен тем же способом, что и s1 (то есть String s2 = "abc";), то обе ссылки s1 и s2 указывали бы на один и тот же объект, так как Java не создает новый экземпляр для строковых литералов, если таковой уже имеется. Вместо этого ссылка будет указывать на уже существующий объект.

13-2. Какие условия должны быть выполнены при переопределении метода equals()?

a.) Переопределение метода equals() предполагает выполнение следующих правил:

1. a) рефлексивность: для любой объектной ссылки x, вызов x.equals(x) возвращает true
2. b) симметричность: для любых объектных ссылок x и y, вызов x.equals(y) возвращает true тогда и только тогда, если вызов y.equals(x) возвращает true
3. c) транзитивность: для любых объектных ссылок x, y и z, если x.equals(y) возвращает true и y.equals(z) возвращает true, тогда вызов x.equals(z) должен вернуть true
4. d) непротиворечивость: для любых объектных ссылок x и y, многократные последовательные вызовы x.equals(y) возвращают одно и то же значение (либо всегда true либо всегда false)
5. e) для любой не равной null объектной ссылки x, вызов x.equals(null) должен вернуть значение false

13-3. Как формально формулируются правила, которым должна следовать реализация метода hashCode()? Как реализован этот метод в классе Object?

a.) Для реализаций метода hashCode() должны быть выполнены следующие правила:

1. a) в одном запуске программы, для одного объекта при вызове метода hashCode(), должно возвращаться одно и то же int значение, если между этими вызовами НЕ были затронуты данные, используемые для проверки объектов на идентичность в методе equals(). Это число НЕ обязано быть одним и тем же при повторном запуске той же программы, даже если все данные будут идентичны
2. b) если два объекта идентичны, то есть вызов метода equals(Object) возвращает true, тогда вызов метода hashCode() у каждого из этих двух объектов должен возвращать одно и то же значение
3. c) если два объекта различны, то есть вызов метода equals(Object) возвращает false, тогда различие их хеш-кодов желательно, но НЕ обязательно. Различие в хеш-кодах для НЕ идентичных объектов нужно только для обеспечения хорошей производительности при использовании этих объектов в хеш-таблицах.

В классе Object метод hashCode() возвращает значение адреса, где хранится объект. Такая реализация удовлетворяет всем правилам и переопределение метода hashCode() в подклассах Object не требуется, если не был переопределен метод equals().

13-4. Если был переопределен метод equals(), какой еще метод, возможно, понадобится переопределить соответствующим образом, чтобы объекты рассматриваемого класса могли быть корректно использованы в хэш-таблицах?

a.) Среди правил, которым должна удовлетворять реализация метода hashCode(), есть следующее: “если два объекта идентичны, то есть вызов метода equals(Object) возвращает true, тогда вызов метода hashCode() у каждого из этих двух объектов должен возвращать одно и то же значение”.

Соответственно, если метод equals() был переопределен, и объекты этого класса планируется использовать в хеш-таблицах, то в соответствии с указанным правилом, метод hashCode() понадобится переопределить таким образом, что бы возвращались одинаковые значения для объектов, вызов метода equals() для которых возвращает true.

13-5. Как реализованы в классе Object методы equals(), toString(), hashCode() ?

a.) В классе Object методы equals(), toString() и hashCode() имеют следующие реализации:

equals() – возвращает true, если ссылки на объекты совпадают

toString() – возвращает строку, которая составляется следующим образом: название класса, символ ‘@’, значение, возвращаемое вызовом метода hashCode(), представленное в шестнадцатеричном виде

hashCode() – имеет native реализацию, возвращающую адрес, по которому хранится объект.

13-6. Какие объекты могут быть клонированы?

a.) Если клонирование производится встроенным методом Object.clone(), то такие классы необходимо специальным образом помечать, указывая, что они реализуют интерфейс Cloneable. Кроме этого, класс может переопределить метод clone() собственным образом, и обойтись без этого интерфейса.

13-7. Какие существуют ограничения на использование метода newInstance() объектов типа Class для создания экземпляров соответствующего класса?

a.) Вызов этого метода создает объект класса, который представляется данным экземпляром Class. Создание будет происходить с помощью конструктора без параметров. Соответственно, что бы создание прошло успешно, такой конструктор должен иметься в классе, а сам класс не должен быть абстрактным.

13-8. Для каких примитивных типов Java существуют классы-обертки? Что будет получено в результате выполнения: (new Integer(1)).equals(new Byte(1)) ?

a.) Классы-обертки существуют для всех примитивных типов Java – byte, short, char, int, long, float, double, boolean (обертка существуют даже для «отсутствия типа» – void).

Вызов (new Integer(1)).equals(new Byte(1)) вернет значение false, так как метод equals() в классах-обертках реализован таким образом, что сначала производится сравнение классов сравниваемых объектов, и если они не совпадают – возвращается значение false.

13-9. В чем особенность класса-обертки для void?

a.) Этот класс, в отличие от остальных классов-оберток, НЕ реализует интерфейс java.io.Serializable. Он не имеет открытого конструктора. Более того, экземпляр этого класса вообще не может быть получен.

13-10. Какие модификаторы присутствуют в определении класса Math? Можно ли от него наследоваться? Можно ли создавать экземпляры этого класса?

a.) Класс Math определен с модификаторами public и final. Соответственно, наследование от класса Math не возможно. Кроме того, единственный конструктор этого класса имеет модификатор доступа private и поэтому может быть вызван только из самого класса Math (впрочем, в классе Math нигде нет вызова этого конструктора). Все методы этого класса – статические.

13-11. В чем проявляется сходство String и примитивных типов Java?

a.) Во-первых, для объектов только этого типа существуют соответствующие литералы, которыми можно инициализировать их значения. Во-вторых, только для одного ссылочного типа String определен оператор “+”.

Кроме того, поскольку класс String является final, String-переменные всегда хранят значения точно такого же типа. Объекты класса String являются

неизменяемыми, поэтому значение переменной не может измениться, если ее передать в качестве аргумента при вызове метода.

13-12. Какой класс используется для представления модифицируемых строк?

a.) Если класс String представляет НЕ модифицируемые строки, то класс StringBuffer позволяет изменять значение строки, которую его объект содержит.

13-13. Какие классы и интерфейсы, необходимые для поддержки многопоточности, определены в пакете java.lang?

a.) Основной класс для обеспечения многопоточности – java.lang.Thread, объекты которого соответствуют потокам. Потоки могут быть объединены в группы потоков, что представляется объектами класса java.lang.ThreadGroup. Создать новый поток можно, унаследовав класс от Thread, либо реализовав интерфейс Runnable.

13-14. Классы каких базовых исключений определены в пакете java.lang ?

a.) Все классы базовых исключений содержатся в пакете java.lang – Throwable, Error, Exception, RuntimeException.