Самарский государственный аэрокосмический университет
имени академика С.П. Королёва

Объектно-ориентированное программирование

Механизмы рефлексии

Занятие 9.1

Самара
2014

План лекции

- Рефлексия и её возможности
- Участники механизма рефлексии
- Получение и работа со ссылкой на описание класса
- Вызов конструкторов и методов
- Управление загрузкой классов

Понятие рефлексии

- Рефлексия (от лат. Reflexio – обращение назад) – обращение субъекта на себя самого, на свое знание или на свое собственное состояние

- Рефлексия в Java – возможность программы анализировать саму себя, взаимодействуя с виртуальной машиной Java (JVM)

Возможности механизма рефлексии

- Загрузка типов во время исполнения программы
- Исследование структуры типов и их элементов
- Создание экземпляров классов
- Вызов методов
- Загрузка классов из набора байтов

Участники механизма рефлексии

- Класс `java.lang.Class`
  - Класс является метаклассом по отношению к другим типам
  - Экземпляры класса `Class` описывают классы и интерфейсы выполняемого приложения
  - Методы класса `Class` позволяют исследовать содержимое описываемого класса и его свойства
- Класс `java.lang.ClassLoader`
  - Реализует механизмы загрузки классов

Участники механизма рефлексии

- Пакет `java.lang.reflect`
  - Содержит ряд дополнительных и вспомогательных классов
  - `Field`

    Описывает поле объекта
  - `Method`

    Описывает метод объекта
  - `Constructor`

    Описывает конструктор объекта
  - `Modifier`

    Инкапсулирует работу с модификаторами
  - `Array`

    Инкапсулирует работу с массивами

Получение представления класса

- Метод `Class Object.getClass()`
  Возвращает ссылку на представление класса, экземпляром которого является объект
- Псевдополе `Object.class`
  Ссылка на представление указанного класса
- Метод `static Class Class.forName(...)`
  Возвращает ссылку на представление класса, полное имя которого указывается параметром типа `String`

Получение представления класса

- Метод `Class[] Class.getClasses()`
  Возвращает ссылку на массив ссылок на объекты `Class` вложенных типов
- Метод `Class Class.getDeclaringClass()`
  Для вложенных типов возвращает ссылку на объект Class внешнего типа
- Метод `Class[] Class.getInterfaces()`
  Возвращает ссылки на описания интерфейсов, от которых наследует тип
- Метод `Class Class.getSuperclass()`
  Возвращает ссылку на описание родительского класса

Пример получения информации о классе

```
import java.lang.reflect.*;

class ListMethods {
  public static void main(String[] argv)
                throws ClassNotFoundException {
    Class c = Class.forName(argv[0]);
    Constructor[] cons = c.getConstructors( );
    printList("Constructors", cons);
    Method[] meths = c.getMethods( );
    printList("Methods", meths);
    Field[] fields = c.getFields();
    printList("Fields", fields);
  }
  static void printList(String s, Object[] o) {
    System.out.println("*** " + s + " ***");
    for (int i = 0; i < o.length; i++)
      System.out.println(o[i].toString( ));
  }
}
```

Возможности класса Class

- Загрузка класса в JVM по его имени
  `static Class forName(String name)`
- Определение вида типа
  `boolean isInterface()`
  `boolean isLocalClass()`
- Получение родительских типов
  `Class getSuperclass()`
  `Class[] getInterfaces()`
- Получение вложенных типов
  `Class[] getClasses()`
- Создание объекта
  `Object newInstance()`

Возможности класса Class

- Получение списка всех полей и конкретного поля по имени
```
Field[] getFields()
Field getField(...)
```
- Получение списка всех методов и конкретного метода по имени и списку типов параметров
```
Method[] getMethods()
Method[] getMethod(...)
```
- Получение списка всех конструкторов и конкретного конструктора по списку типов параметров
```
Constructor[] getConstructors()
Constructor getConstructor(...)
```

Передача параметров в методы

- Поскольку на момент написания программы типы и даже количество параметров неизвестно, используется другой подход:
  - Ссылки на все параметры в порядке их следования помещаются в массив типа `Object`
  - Если параметр имеет примитивный тип, то в массив помещается ссылка на экземпляр класса-оболочки соответствующего типа, содержащий необходимое значение
- Возвращается всегда тип `Object`
  - Для ссылочного типа используется приведение типа или рефлексивное исследование
  - Для примитивных типов возвращается ссылка на экземпляр класса-оболочки, содержащий возвращенное значение

Создание экземпляров классов

- Метод `Object Class.newInstance()`
  Возвращает ссылку на новый экземпляр класса, используется конструктор по умолчанию

- Метод
  ```
  Object Constructor.newInstance(
      Object[] initArgs)
  ```
  Возвращает ссылку на новый экземпляр класса, с использованием конструктора и указанными параметрами конструктора

Вызов методов

- Прямой вызов
  - Если на момент написания кода известен тип-предок загружаемого класса
  - Приведение типа и вызов метода

- Вызов через экземпляр класса `Method`
  `Object Method.invoke(Object obj, Object[] args)`
  - `obj` – ссылка объект, у которого должен быть вызван метод
  - принято передавать `null`, если метод статический
  - `args` – список параметров для вызова методов

Пример вызова статического метода

```
import java.lang.reflect.*;
public class Main {
  public static void main(String[] args) {
    if (args.length == 3) {
      try {
        Class c = Class.forName(args[0]);
        Method m = c.getMethod(args[1], new Class [] {Double.TYPE});
        Double val = Double.valueOf(args[2]);
        Object res = m.invoke(null, new Object [] {val});
        System.out.println(res.toString());
      } catch (ClassNotFoundException e) {
        System.out.println("Класс не найден");
      } catch (NoSuchMethodException e) {
        System.out.println("Метод не найден");
      } catch (IllegalAccessException e) {
        System.out.println("Метод недоступен");
      } catch (InvocationTargetException e) {
        System.out.println("При вызове возникло исключение");
      }
    }
}}
```

Класс ClassLoader

- Экземпляры класса отвечают за загрузку классов в виртуальную машину
- Это абстрактный класс, не имеющий ни одного абстрактного метода

- Классы-наследники должны в каком-то смысле расширять возможности виртуальной машины по загрузке классов
- Объекты загрузчиков образуют иерархию (родительский объект указывается как параметр защищённых конструкторов)

Основные методы класса ClassLoader

- `public Class loadClass(String name)`
  - Проверяет, не был ли класс загружен раньше
  - Вызывает аналогичный метод родительского объекта
  - Вызывает метод `findClass()`, чтобы найти класс
  - Не стоит переопределять этот метод
- `protected Class findClass(String name)`
  - Ищет и загружает класс по имени специфическим для данного загрузчика способом
  - Этот метод и нужно переопределять
- `protected final Class defineClass( String name, byte[] b, int off, int len)`
  - Загружает класс из указанного набора байтов

Пример использования загрузчика классов

```
package reflectiontest;

import java.io.*;

public class ArbitraryFileLoader extends ClassLoader {
  public Class loadClassFromFile(String filename) throws IOException {
    byte[] b = loadClassData(filename);
    return defineClass(null, b, 0, b.length);
  }

  private byte[] loadClassData(String filename) throws IOException {
    FileInputStream in = new FileInputStream(filename);
    byte[] fileContent = new byte[in.available()];
    in.read(fileContent);
    in.close();
    return fileContent;
  }
}
```

Пример использования загрузчика классов

```
public class ReflectionTest {
    public static void main(String[] args) {
        try {
            ArbitraryFileLoader l = new ArbitraryFileLoader();
            Object o = l.loadClassFromFile(args[0]).newInstance();
            if (o instanceof Somethingable) {
                System.out.println(((Somethingable) o).something());
            }
        } catch (java.io.IOException ex) {
            System.out.println("Problem with reading of file");
        } catch (ClassFormatError ex) {
            System.out.println("File doesn't contain a class");
        } catch (IllegalAccessException ex) {
            System.out.println("No public constructor");
        } catch (InstantiationException ex) {
            System.out.println("Problem during object creation");
        }
    }
}
```

Самарский государственный аэрокосмический университет
имени академика С.П. Королёва

Объектно-ориентированное программирование

Нововведения Java

Занятие 9.2

Самара
2014

План лекции

- Статический импорт
- Автоупаковка/автораспаковка
- Переменное количество аргументов в методах
- Параметризованные типы
- Цикл for-each
- Перечислимые типы

Проблема

- Имеется:

```
hypot = Math.sqrt(Math.pow(side1, 2)
                  + Math.pow(side2, 2));
```

- Хотелось бы:

```
hypot = sqrt(pow(side1, 2)
             + pow(side2, 2));
```

Статический импорт

- Импорт элемента типа

```
import static pkg.TypeName.staticMemberName;
```

```
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
```

- Импорт всех элементов типа

```
import static pkg.TypeName.*;
```

```
import static java.lang.Math.*;
```

Особенности статического импорта

- Повышает удобство написания программ и уменьшает объем кода
- Уменьшает удобство чтения программ
- Приводит к конфликтам имен
- Мораль: рекомендуется к использованию только когда действительно необходим

Проблема

- Имеется:

```
List list = new LinkedList();
list.add(new Integer(1));
list.add(new Integer(10));
```

- Хотелось бы:

```
List list = new LinkedList();
list.add(1);
list.add(10);
```

Автоупаковка и авторапаковка

- **Автоупаковка** – процесс автоматической инкапсуляции данных простого типа в экземпляр соответствующего ему класса-обертки в случаях, когда требуется значение ссылочного типа
- **Авторапаковка** – процесс автоматического извлечения примитивного значения из объекта-упаковки в случаях, когда требуется значение примитивного типа

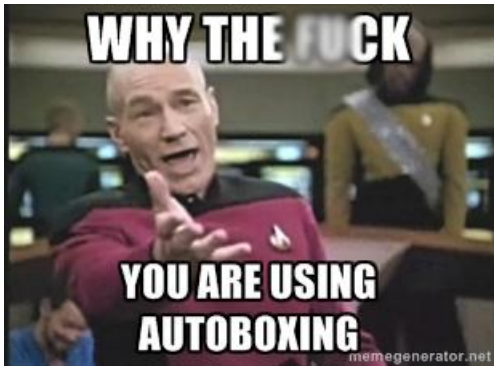
```
List list = new LinkedList();
list.add(1);
int b = (Integer)list.get(0) + 10;
```

Особенности автоупаковки

- Происходит при присваивании, вычислении выражений и при передаче параметров
- Объекты создаются без использования ключевого слова **new**

```
Integer i = 15;
```

- Объекты создаются!
- Вообще полагать, что примитивные типы не нужны
- Автоупаковка требует существенных ресурсов
- Злоупотреблять автоупаковкой вообще не стоит!



Проблема

- Имеется:

```
int s1 = sum(new int[] {1, 2});
int s2 = sum(new int[] {1, 2, 3});
```

- Хотелось бы:

```
int s1 = sum(1, 2);
int s2 = sum(1, 2, 3, 4);
```

Переменное количество аргументов

- Пример метода

```
int sum(int ... a) {
  int s = 0;
  for (int i = 0; i < a.length; i++)
    s += a[i];
}
```

- Пример вызова

```
int s2 = sum(1, 2, 3);
int s1 = sum(new int[] {1, 2});
```

Особенности переменного количества аргументов

- Внутри там все равно живет массив…
- Аргумент переменной длинны в методе может быть только один

- Аргумент переменной длинны должен быть последним в списке аргументов метода
- В сочетании с перегрузкой методов способен приводить к изумительным ошибкам компиляции в виду неоднозначности кода

Проблема

- Имеется:

```
List list= new LinkedList ();
list.add(10);
list.add(5);
list.add((Integer)list.get(0) + (Integer)list.get(1));
```

- Хотелось бы:

```
List<Integer> list= new LinkedList<Integer>();
list.add(10);
list.add(5);
list.add(list.get(0) + list.get(1));
```

Параметризованные типы

- **Параметризованные типы (настраиваемые типы, generic types)**
- Позволяют создавать классы, интерфейсы и методы, в которых тип обрабатываемых данных задается как параметр
- Позволяют создавать более компактный код, чем универсальные (обобщенные) типы, использующие ссылки типа `Object`
- Обеспечивают автоматическую проверку и приведение типов
- Позволяют создавать хороший, годный повторно используемый код

Скромный пример

- Пример класса

```
class Generic<T> {
  T obj;
  Generic(T o) {obj = o;}
  T getObj() {return obj;};
}
```

- Пример использования

```
Generic<Integer> iObj;
iObj = new Generic<Integer>(33);
int i = iObj.getObj() + 10;
```

Особенности параметризованных типов

- Использовать примитивные типы в качестве параметров-типов нельзя
- Если одинаковые настраиваемые типы имеют различные аргументы, то это различные типы
- Обеспечивается более жесткий контроль типов на стадии компиляции

Общий синтаксис

- Объявление настраиваемого типа

  `class имяКласса<список-формальных-параметров> {...}`

```
class Generic2<T, E> {...}
```

- Создание ссылки и объекта настраиваемого типа

  `имяКласса<список-фактических-параметров> имяПеременной = new имяКласса<список-фактических-параметров>(список-аргументов);`

```
Generic2<Integer, String> gObj = new
    Generic2<Integer, String>(10, "ok");
```

Ограниченные типы

```
class Stats<T extends Number> {
  T[] nums;
  Stats(T[] o) {nums = o;}
  double average() {
    double sum = 0.0;
    for(int i = 0; i < nums.length; i++)
      sum += nums[i].doubleValue();
    return sum / nums.length;
  }
}
```

- Ограничение типа позволяет использовать у ссылок методы и поля, доступные в типе-ограничителе
- Типы, не наследующие от указанного, не могут быть использованы при создании объектов
- Как имя типа может быть указан интерфейс!!!
- Как имя типа может быть указан ранее введенный параметр!!!

```
class Generic3<T extends Comparable<T>> {...}
```

Метасимвольный аргумент

- Что делать при передаче экземпляров параметризованных типов в методы, т.е. как писать сигнатуру?
- Для этого используется метасимвол, обозначающий произвольный тип-параметр

```
class Generic<T> {
  ...
  boolean compare(Generic<?> o) {
    return o.getObj() == obj;
}
```

Метасимвол с ограничениями

- Ограничение сверху

  `<? extends super>`

  Тип ***super*** допускается

- Ограничение снизу

  `<? super sub>`

  Тип ***sub*** не допускается

Параметризованные методы

- Методы могут иметь собственные типы-параметры
- Фактические аргументы, передаваемые в формальные аргументы, имеющие тип-параметр, будут проверяться на соответствие типу, причем на этапе компиляции
- Пример метода

```
public static <T extends Comparable<T>> T min(T[] v) {
  T min = v[0];
  for (int i = 1; i < v.length; i++)
      if (min.compareTo(v[i]) > 0)
        min = v[i];
  return min;
}
```

- Пример использования

```
System.out.println(min(new Integer[] {10, 15, 5}));
```

Ряд особенностей

- Конструкторы могут быть параметризованными (даже если сам класс таковым не является)
- Интерфейсы могут быть параметризованными
- Нельзя создавать объекты, используя типы-параметры
- Статические члены класса не могут использовать его типы-параметры
- Настраиваемый класс не может расширять класс `Throwable`
- От настраиваемых типов можно наследовать, есть ряд особенностей

Ряд особенностей

- Нельзя создать массив типа-параметра

```
class Generic<T> {
  T[] vals; //OK

  Generic(T[] nums) {
    //vals = new T[10]; //Не есть правильно!
    vals = nums; //OK
  }
}
```

- Массивов элементов конкретной версии параметризованного типа не бывает

```
//Generic<Integer>[] gens = ew Generic<Integer>[10];//Nicht OK
Generic<?>[] gens = new Generic<?>[10];
```

И как же это работает?

- Механизм стирания
  - В реальном байт-коде никаких настраиваемых типов в целом-то и нет…
  - Информация о настраиваемых типах удаляется на стадии компиляции
  - Именно компилятор осуществляет контроль безопасности приведения типов
  - А внутри после компиляции все те же «обобщенные» классы, явные приведения типов и прочее, и прочее…

Ошибки неоднозначности

- «Логически правильный» код

```
public class Test <T> {
  T first(T[] arr) {
    return arr[0];
  }
  Object first(Object[] arr) {
    return arr[0];
  }
}
```

- Оказывается неверным с точки зрения компилятора

```
Test.java:6: first(T[]) is already defined in Test
  Object first(Object[] arr) {
         ^
```

- И это – самый простой пример…

Проблема

- Имеется:

```
int[] nums = {1, 2, 3, 4, 5};
int sum = 0;
for (int i = 0; i < 5; i++)
    sum += nums[i];
```

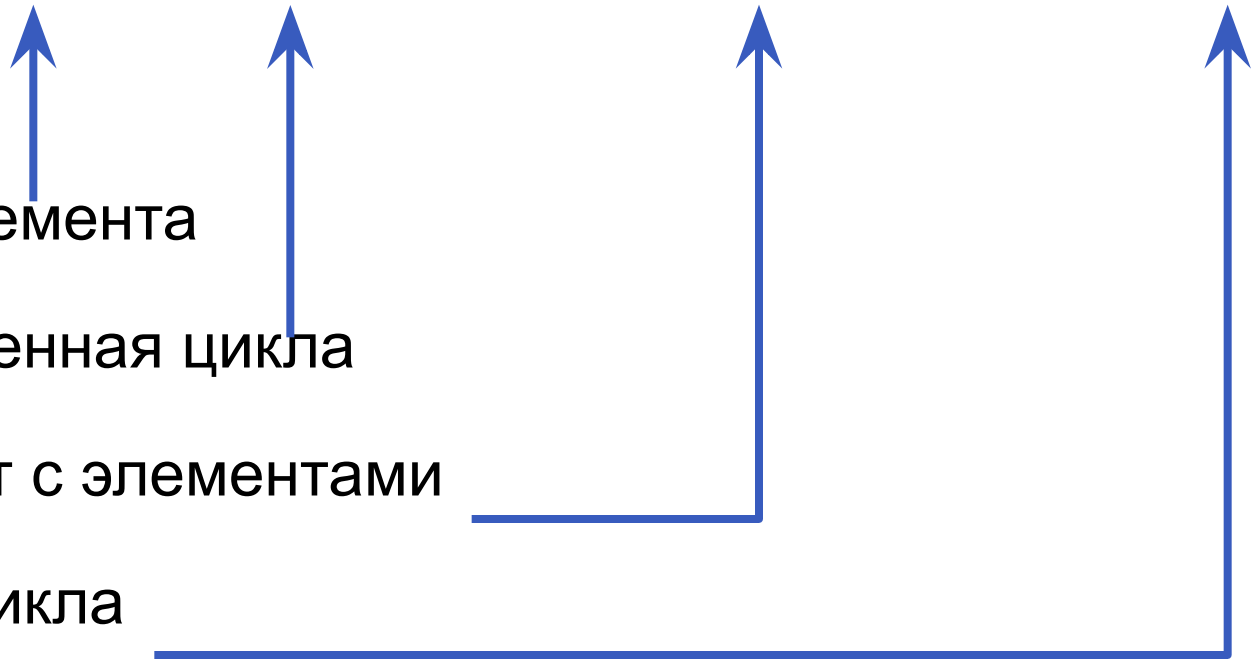
- Хотелось бы:

```
int[] nums = {1, 2, 3, 4, 5};
int sum = 0;
for (int x: nums)
    sum += x;
```

Улучшенный цикл for (for-each)

- Общая форма записи

```
for (type iterVar : iterableObj) statement;
```



- Тип элемента
- Переменная цикла
- Агрегат с элементами
- Тело цикла

Работа улучшенного цикла for

- В каждом витке цикла «извлекается» очередной элемент агрегата
- Ссылка на него (для ссылочных типов) или значение (для примитивных) помещается в переменную цикла
- Тип переменной цикла должен допускать присвоение элементов агрегата
- Цикл выполняется до тех пор, пока не будут перебраны все элементы агрегата

Пример обработки многомерных массивов

```
int sum = 0;
int nums[][] = new int[3][5];

for (int i = 0; i < 3; i++)
  for (int j = 0; j < 5; j++)
    nums[i][j] = (i + 1) * (j + 1);

for (int[] x: nums)
  for (int y: x)
    sum += y;
```

Особенности улучшенного цикла for

- По сути это внутренний итератор
- Переменная цикла доступна только для чтения…
- Порядок обхода в целом не определен…
- Нет доступа к соседним элементам…
- Мораль:
  - Область применения обобщенного цикла for «несколько уже», чем у «необобщенной» версии
  - Зато для этого класса задач синтаксис обобщенного цикла существенно удобнее

Внимание, вопрос!

- А кто же управляет итерациями?
- Агрегат обязан реализовывать интерфейс `java.lang.Iterable<T>`
- Сей интерфейс содержит лишь один элемент `Iterator<T> iterator()`
- Данный, вроде бы знакомый, интерфейс, тоже претерпел некоторые изменения:
  - `boolean hasNext()`
  - `void remove()`
  - `T next()`

Проблема

- Имеется:

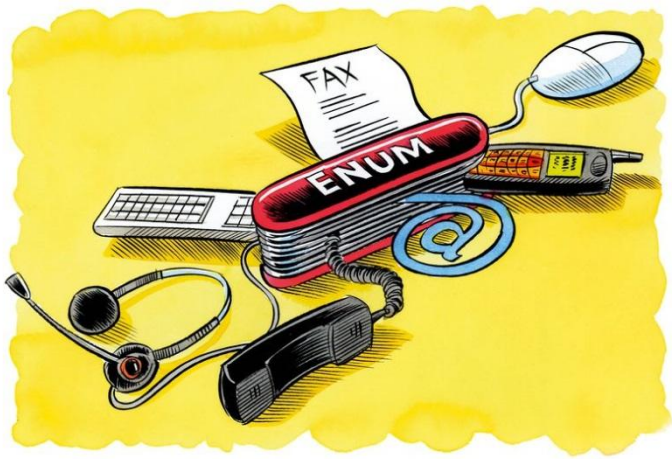
```
class Apple {
  public static final int JONATHAN = 0;
  public static final int GOLDENDEL = 1;
  public static final int REDDEL = 2;
  public static final int WINESAP = 3;
  public static final int CORTLAND = 4;
}
```

- Хотелось бы:

```
enum Apple {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
```

Перечислимые типы

- Перечислимый тип
  `Apple`
- Константы перечислимого типа
  `Jonathan`, `GoldenDel`, `RedDel`…
- Объявление переменной




```
Apple ap;
```

- Присвоение переменной значения

```
ap = Apple.RedDel;
```

Перечислимые типы

- Проверка равенства

```
if(ap == Apple.GoldenDel)
```

- Использование в блоке переключателей

```
switch(ap) {
  case Jonathan: //...
  case Winsap: //...
  default: //...
}
```

А теперь отличия от классики

- Перечислимый тип – это класс!
- Да к тому же имеет методы!
  - `public static enumType[] values()` возвращает ссылку на массив ссылок на все константы перечислимого типа

```
Apple[] allApples = Apple.values();
```

  - `public static enumType valueOf(String str)` возвращает константу перечислимого типа, имя которой соответствует указанной строке, иначе выбрасывает исключение

```
Apple ap = Apple.valueOf("Jonathan");
```

И еще отличия…

- Можно определять конструкторы, добавлять поля и методы, реализовывать интерфейсы

```
enum Apple {
  Jonathan(10), GoldenDel(9), RedDel, Winsap(15), Cortland(8);
  private int price;
  Apple(int p) {
    price = p;
  }
  Apple() {
    price = -1;
  }
  int getPrice() {
    return price;
  }
}
```

Особенности перечислимых типов

- Создавать экземпляры с помощью оператора `new` нельзя!
- Все перечислимые типы наследуют от класса `java.lang.Enum`
- Клонировать экземпляры нельзя, сравнивать и выполнять прочие стандартные операции – можно

Спасибо за внимание!

Дополнительные источники

- Арнолд, К. Язык программирования Java [Текст] / Кен Арнолд, Джеймс Гослинг, Дэвид Холмс. – М. : Издательский дом «Вильямс», 2001. – 624 с.
- Вязовик, Н.А. Программирование на Java. Курс лекций [Текст] / Н.А. Вязовик. – М. : Интернет-университет информационных технологий, 2003. – 592 с.
- Эккель, Б. Философия Java [Текст] / Брюс Эккель. – СПб. : Питер, 2011. – 640 с.
- Шилдт, Г. Java 2, v5.0 (Tiger). Новые возможности [Текст] / Герберт Шилдт. – СПб. : БХВ-Петербург, 2005. – 206 с.
- JavaSE at a Glance [Электронный ресурс]. – Режим доступа: http://www.oracle.com/technetwork/java/javase/overview/index.html, дата доступа: 21.10.2011.
- JavaSE APIs & Documentation [Электронный ресурс]. – Режим доступа: http://www.oracle.com/technetwork/java/javase/documentation/api-jsp-136079.html, дата доступа: 21.10.2011.