Лекция 8

Класс Modifier

Класс Modifier кодирует все модификаторы, используемые в объявлениях типа, в виде констант:

ABSTRACT, FINAL, INTERFACE, NATIVE, PRIVATE, PROTECTED, PUBLIC, STATIC, STRICT, SYBCHRONIZED, TRANSIDENT, VOLATILE.

Каждой из констант отвечает метод запрос вида **isMod(int modifier)** (тут Mod одно из выше приведенных имен, например, isPublic), который возвращает true, если модификатор mod присутствует в объявлении типа.

Рассмотрим пример. Пусть имеется объявление поля

public static final int s=10;

тогда значение возвращаемое методом getModifiers соответствующего объекта класса Field будет иметь вид

Modifier.PUBLIC | Modifier.STATIC | Modifier.FINAL

Модификатор strictfp представляется константой STRICT.

Методы- запросы можно использовать в следующей форме

Modifier.isPrivate(field.getModifiers());

это эквивалентно следующему условию

(field.getModifiers()&Modifier.PRIVATE)!=0

Класс Field

В составе класса Field реализованы методы, позволяющие запрашивать информацию о типе поля, а также считывать и задавать его значение.

Рассмотрим некоторые методы класса Field

1. **getType()** – возвращает объект класса Class, отвечающий типу текущего поля. Например, для поля типа int, получим int.class.

2. **Методы set и get** – позволяют считывать текущее значение поля, а также задавать новое. Рассмотрим пример:

```
public static void printField(Object o,
                              String name) throws
                               NoSuchFieldException,
                               IllegalAccessException{
   Field field = o.getClass().getField(name);
   Short value = (Short) field.get(o);
   System.out.println(value);
}
```

Т.е. метод get возвращает значение, на которое ссылается соответствующее поле или объект класса –оболочки.

Пример использования метода set имеет вид:

```
public static void setField(Object o, String name,
                                        short nv) throws
                                NoSuchFieldException,
                                IllegalAccessException{
   Field field = o.getClass().getField(name) ;
   field.set(o,new Short(nv));
}
```

Для сохранения nv в поле данного объекта необходимо использовать классы оболочки.

Существуют также методы имеющие вид **getPrimitiveType** (например, getInt) и **setPrimitiveType**. Эти методы можно использовать для изменения полей в классе, имеющих примитивный тип. Например,

```
field.setShort(o,nv);
```

Класс Method

Средства класса Method - позволяют получать полную информацию, касающуюся объявлений методов определенного класса, и при необходимости вызывать эти методы в контексте заданных объектов.

Рассмотрим методы класса Method.

1. **public Class getReturnType()** - возвращает объект Class, соответствующий типу значения, возвращаемого текущим методом. Если вместо типа возвращаемого значения в объявлении метода указано служебное слово void, рассматриваемый метод вернет объект void.class.

2. **public Class[] getParameterTypes()** - возвращает массив объектов Class, соответствующих типам параметров, которые заданы в объявлении текущего метода. Объекты заносятся в массив в том порядке, в каком параметры перечислены в объявлении метода. Если метод не имеет параметров, возвращается пустой массив.
3. **public Class[] getExceptionTypes()** - возвращает массив объектов Class, соответствующих типам исключений, которые заданы в предложении throws объявления текущего метода. Объекты заносятся в массив в том порядке, в каком наименования типов исключений перечислены в объявлении метода.

4. **public Object invoke(Object onThis, Object[] args)**
throws IllegalAccessException,
IllegalArgumentException,
InvocationTargetException

Вызывает метод, определяемый текущим объектом Method, в контексте объекта onThis с заданием значений аргументов, передаваемых массивом args.

Для нестатических методов выбор реализации осуществляется на основании фактического типа объекта, определяемого параметром onThis. Для статических методов onThis не принимается во внимание и может быть равен null.

Длина массива args должна совпадать с числом параметров в объявлении метода, а типы объектов-элементов массива должны допускать присваивание соответствующим типам параметров метода — в противном случае будет выброшено исключение IllegalArgumentException.

Если в составе объекта, определяемого параметром onThis, нет типа, членом которого является текущий метод, выбрасывается исключение IllegalArgumentException.

Если значение onThis равно null и метод не статический, генерируется исключение типа NullPointerException.

Если выполнение вызываемого метода завершается аварийно, выбрасывается исключение типа InvocationTargetException.

Рассмотрим пример. Вызовем средствами рефлексии метод **return str.indexOf(".", 8)**

тогда имеем

```
try {
  Class strClass = str.getClass();
  Method indexM = strClass.getMethod("indexOf",
                    new Class[] { string.class,  int.class } );
  Object  result = indexM.invoke(str, new object[] {
                                                ".",  new Integer(8)} );
  return  ((Integer) result).intValue();
}
catch  (NoSuchMethodException e)  { …….. }
catch  (invocationTargetException e) {…….. }
catch  (illegalAccessException e)  {……}
```

Класс Constructor

Для создания новых экземпляров (объектов) типа может быть использован метод **newInstance объекта Class**, соответствующего этому типу.

Метод вызывает конструктор без аргументов, принадлежащий типу, и возвращает ссылку на вновь созданный объект класса Object, который должен быть явно преобразован к требуемому типу.

Рассмотрим пример.

```
static double[]  testData = { 0.3,1.3e-2, 7.9, 3.17 };
 public static void main(String[] args){
  try {
    for(int arg = 0; arg < args.length; arg++){
        String name = args[arg];
        Class classFor = Class.forName(name);
        SortDouble sorter =
                  (SortDouble)classFor.newInstance();
        SortMetrics metrics = sorter.sort(testData);
        System.out.println(name + ": " + metrics);
    for(int i  =0;  i  < testData.length; i++)
        System.out.println(“ " + testData[i]); } }
 catch(Exception e) { System.err.println(e); } }
```

Метод **newInstance** при некорректном применении способен выбрасывать большое число объектов исключений различных типов.

InstantiationException-если класс, объект которого должен быть создан, не обладает конструктором без аргументов, либо определен как абстрактный, либо в действительности является интерфейсом, либо выполнение процедуры создания объекта прерывается по каким-либо иным причинам.

IllegalAccessException - если класс либо его конструктор без аргументов недоступны.

SecurityException - если действующая политика безопасности запрещает создание новых объектов

ExceptionInInitializerError –выбрасывается при инициализации класса.

В классе Constructor определены и другие методы.

public Class[] getParameterTypes()

Возвращает массив объектов Class, соответствующих типам параметров, которые заданы в объявлении текущего конструктора.

public Class[] getExceptionTypes()

Возвращает массив объектов Class, соответствующих типам исключений, которые заданы в предложении throws объявления текущего конструктора.

public Object newInstance(Object[] args)
throws InstantiationException,
IllegalAccessException,
IllegalArgumentException,
InvocationTargetException

Использует конструктор, представляемый текущим объектом Constructor, для создания и инициализации нового экземпляра класса, в котором конструктор объявлен, с передачей заданных аргументов.

Возвращает ссылку на вновь созданный и инициализированный объект. Длина массива args должна совпадать с числом параметров в объявлении конструктора, а типы объектов-элементов массива должны допускать присваивание соответствующим типам параметров конструктора — в противном случае будет выброшено исключение IllegalArgumentException.

Рассмотрим пример:

```
class Myclass {
    private int a;
    public Myclass(int k){a=k;}
    public int func(int a,int b){return a+b;}
}
public class Main{
 public static void main(String[] args){
    try{
      String name="Myclass";
      Class mycl=Class.forName(name);
      Class[] d={int.class};
      Constructor c=mycl.getConstructor(d);
      Myclass ob=(Myclass)c.newInstance(new Object[]{
                                                    new
    Integer(10)});
       System.out.println(ob.func(3,5));  }
    catch(Exception e){};
}}
```

Класс AccessibleObject

Классы Field, Constructor и Method являются производными от класса AccessibleObject, который дает возможность разрешать или запрещать проверку признаков доступа уровня языка, таких как public и private.

Класс AccessibleObject имеет методы

1. **public void setAccessible(boolean flag)**

Устанавливает флаг доступа к объекту в соответствии со значением аргумента: true означает, что объект больше не подчиняется правилам доступа, устанавливаемым на уровне языка (и будет всегда доступен), а false вынуждает объект поддерживать заданный уровень доступа.

Если полномочий по изменению флага доступа недостаточно, выбрасывается исключение типа SecurityException

2. public static
void setAccessible(AccessibleObject[] array, boolean flag)

Позволяет установить флаг доступа к объектам, передаваемым в виде массива.

Если в процессе обработки очередного объекта выбрасывается исключение типа SecurityException, объекты, расположенные в массиве ранее, сохраняют вновь заданные значения уровня доступа, а все остальные объекты остаются в прежнем состоянии.

3. public boolean isAccessible()

Возвращает текущее значение флага доступа к объекту

Класс Array

Класс Array используется для создания массива средствами рефлексии.

Для создания массивов используются две формы метода newInstance.

public Object newInstance(Class compType, int length)

Возвращает ссылку на новый массив типа compType заданной длины length.

public Object newInstance(Class compType, int[] dim)

Возвращает ссылку на новый многомерный массив типа compType, размерности которого заданы значениями элементов массива-параметра dim.

Если массив dim пуст или обладает длиной, превышающей допустимое число размерностей (обычно 255), выбрасывается исключение типа llegalArgumentException.

Рассмотрим примеры.

Пример1. Сформируем массив типа byte

```
byte[] ba =  (byte[])
                    Array.newInstance(byte.class,13);
```

Это равнозначно

```
  byte[] ba = new byte[13];
```

Пример2.

```
 int[] dims = {4, 4};
 double[][] matrix =(double[][])
                Array.newInstance(double.class, dims);
```

Это равнозначно

```
double[][]  matrix = new double[4][4];
```

Класс Array обладает методами get и set.

Пусть задан массив xa значений типа int; тогда выражению xa[i] будет соответствовать:

Integer n=Array.get(xa, i)

Присвоить значение элементу массива можно так: xa[i] = 23; - это то же самое, что

Array.set(xa, i, new Integer(23));

Класс Package

Вызов метода **getPackage** класса Class позволяет получить объект класса Package, содержащий описание пакета, в составе которого определен класс (сам класс Package размещен в пакете java.lang).

Метод **getName()** объекта Package возвращает полное имя текущего пакета.

Класс Proxy

Класс Proxy позволяет динамически создавать классы, реализующие один или несколько интерфейсов.

Предположим, что имеется класс *A*, реализующий некоторые интерфейсы.

Java-машина во время исполнения может сгенерировать прокси-класс для данного класса *A*, т.е. такой класс, который реализует все интерфейсы класса *A*, но заменяет вызов всех методов этих интерфейсов на вызов метода **invoke, интерфейса InvocationHandler**,для которого можно определять свои реализации.

Создается прокси-класс с помощью вызова метода **Proxy.getProxyClass**, который принимает ClassLoader и массив интерфейсов (interfaces), а возвращает объект класса java.lang.Class, который загружен с помощью переданного ClassLoader и реализует переданный массив интерфейсов.

На передаваемые параметры есть ряд ограничений:

1. Все объекты в массиве interfaces должны быть интерфейсами. Они не могут быть классами или примитивами.

2. В массиве interfaces не может быть двух одинаковых объектов.

3. Все интерфейсы в массиве interfaces должны быть загружены тем ClassLoader, который передается в метод getProxyClass.

4. Все не публичные интерфейсы должны быть определены в одном и том же пакете, иначе генерируемый прокси-класс не сможет их все реализовать.

5. Ни в каких двух интерфейсах не может быть метода с одинаковым названием и сигнатурой параметров, но с разными типами возвращаемого значения.

6. Длина массива interfaces ограничена 65535-ю интерфейсами. Никакой Java-класс не может реализовывать более 65535 интерфейсов.

Свойства динамического прокси-класса

1. Прокси-класс является публичным, снабжен модификатором final и не является абстрактным.

2. Имя прокси-класса по-умолчанию не определено, однако начинается на Proxy. Все пространство имен, начинающихся на Proxy зарезервировано для прокси-классов

(В последних версиях Java это не обязательно).

3. Прокси-класс наследуется от

java.lang.reflect.Proxy.

4. Прокси-класс реализует все интерфейсы, переданные при создании, в порядке передачи.

5. Если прокси-класс реализует непубличный интерфейс, то он будет сгенерирован в том пакете, в котором определен этот самый непубличный интерфейс. В общем случае пакет, в котором будет сгенерирован прокси-класс неопределен.

6. Метод **Proxy.isProxyClass** возвращает true для классов, созданных с помощью Proxy.getProxyClass и для классов объектов, созданных с помощью **Proxy.newProxyInstance** и false в противном случае.

Данный метод используется подсистемой безопасности Java и нужно понимать, что для класса, просто унаследованного от java.lang.reflect.Proxy он вернет false.

Свойства созданного экземпляра прокси-класса следующие:

1. Объект прокси-класса приводим ко всем интерфейсам, переданным в массиве interfaces. Если IDemo - один из переданных интерфейсов, то операция proxy instanceof IDemo всегда вернет true, а операция (IDemo) proxy завершится корректно.

   2. Статический метод **Proxy.getInvocationHandler** возвращает обработчик вызовов, переданный при создании экземпляра прокси-класса. Если переданный в данный метод объект не является экземпляром прокси-класса, то будет выброшено IllegalArgumentException исключение.

   3. Класс-обработчик вызовов реализует интерфейс InvocationHandler, в котором определен метод invoke, имеющий следующую сигнатуру:

   public Object invoke(Object proxy, Method method,
   Object[] args) throws Throwable

Рассмотрим пример:

```
package javaapplication3;

interface Account {
  double getBalance();
  void changeBalance(int sum);
  void percents(double per);}

class MyAccount implements Account{
    private double balance;
    public MyAccount(){ balance=0.0; }
    public double getBalance(){ return balance; }
    public void changeBalance(int sum){
            balance+=sum;}
 public void percents(double per){
          balance+=balance*per/100; }; }
```

```
class MyAccountProxy implements
                                        InvocationHandler{
  private Account ac;
  public MyAccountProxy(Account acc){ ac=acc; }
  public static Account newInstance(Account da){
   return (Account)Proxy.newProxyInstance(
                    da.getClass().getClassLoader(),
                    da.getClass().getInterfaces(),
                     new MyAccountProxy(da));
}
```

```
public Object invoke(Object proxy,
                     Method method, Object[] args)
                                      throws Throwable{
    if(method.getName()=="percents"){
       double d=((Double)args[0]).doubleValue();
       if (d<0) d=0;
       if(d>30) d=30;
       args[0]=new Double(d);
       return method.invoke(ac, args);  }
     else{
       return method.invoke(ac, args); }
     }
}
```

```
public class Main{
 public static void main(String[] args){
    MyAccount ma=new MyAccount();
  Account
   a=(Account)MyAccountProxy.newInstance(ma);
     a.changeBalance(150);
     System.out.println(a.getBalance());
    a.percents(20);
     System.out.println(a.getBalance());
     a.percents(35);
    System.out.println(a.getBalance());} }
```

Загрузка классов

Исполняющая система загружает классы по мере возникновения необходимости в них. Функциональные особенности процедур загрузки классов существенным образом зависят от реализации виртуальных машин Java, но в большинстве случаев для отыскания классов, адресуемых приложением, но не загруженных исполняющей системой, применяется механизм просмотра пути поиска классов.

Чтобы создать приложение, которое в состоянии загружать классы способами, отличными от предусмотренных по умолчанию, следует воспользоваться объектом класса ClassLoader, способным получить байт-код реализации нужного класса и загрузить его в среду исполняющей системы.

Класс **ClassLoader** является абстрактным классом. Для создания собственного загрузчика классов, необходимо создать класс – наследник от ClassLoader и переопределить метод

protected Class findClass(String name) throws ClassNotFoundException

Который находит байт-код класса с заданным именем **name** и загружает данные в среду виртуальной машины, возвращая объект Class, представляющий найденный класс.

Объект-загрузчик способен делегировать полномочия по загрузке классов "родительскому" загрузчику классов (parent class loader). "Родительский" загрузчик классов может быть задан в качестве аргумента конструктора класса ClassLoader.

protected ClassLoader()

Создает объект ClassLoader, неявно используя в качестве "родительского" загрузчика классов системный загрузчик (который может быть получен посредством вызова метода **getSystemClassLoader**).

protected ClassLoader(ClassLoader parent)

Создает объект ClassLoader, используя заданный "родительский" загрузчик классов.

Основным в составе класса ClassLoader является метод loadClass

public Class loadClass(String name) throws ClassNotFoundException

возвращает объект Class для класса с заданным именем и при необходимости загружает этот класс. Если класс не может быть загружен, выбрасывается исключение типа ClassNotFoundException.

Схема загрузки классов, предлагаемая методом **loadClass** по умолчанию и обычно не переопределяемая, выглядит так:

1. проверить посредством вызова метода findLoadedClass класса ClassLoader, не загружался ли заданный класс раньше; в составе ClassLoader предусмотрена таблица объектов Class для всех классов, загруженных средствами текущего загрузчика классов; если класс был загружен прежде, метод findLoadedClass возвратит ссылку на существующий объект Class;

2. если класс не загружался, вызывается loadClass "родительского" загрузчика классов; если текущий загрузчик не обладает "родителем", используется системный загрузчик классов;
3. если класс все еще не загружен, вызывается метод findClass, выполняющий поиск и загрузку класса.

Таким образом, необходимо реализовать собственные версии следующих методов ClassLoader:

- **protected synchronized Class**
  loadClass(String name,boolean resolve)
  throws ClassNotFoundException

- **protected Class findClass(String name)**
  throws ClassNotFoundException

- **protected java.net.URL findResource(String name)**

- **protected java.util.Enumeration**
  findResources(String name) throws IOException

(Абстрактный класс ClassLoader представляет только реализацию метода loadClass, основанную на protected-методах – findLoadedClass и findClass**).**

Рассмотрим пример.

```
class  PlayerLoader extends ClassLoader {
   public Class findClass(String name) throws
                                        ClassNotFoundException {
      try {
         byte[] buf = bytesForClass(name);
         return defineClass(name, buf, 0, buf.length);
      }
      catch  (IOException e)  {
          throw new ClassNotFoundException(e.toString());
      }
   }
  //  ... Объявления метода bytesForClass и других
                                                    методов
}
```

Метод findClass обычно выполняет две функции.

Во-первых, он должен обнаружить байт-код заданного класса и сохранить его в массиве типа byte — эта обязанность в примере возложена на метод bytesForClass.

Во-вторых, он использует прикладной метод defineClass, чтобы выполнить фактическую загрузку класса, определяемого байт-кодом.

Метод defineClass имеет вид

protected final Class defineClass(String name, byte[] data, int offset, int length) throws ClassFormatError

Возвращает объект Class для класса с заданным именем **name**; бинарное представление класса передается в виде массива **data**.

Для загрузки класса используются только байты, содержащиеся в элементах массива **data** с индексами от **offset** до **offset+length**. Если байты из указанного промежутка не удовлетворяют требуемому формату описания класса, выбрасывается объект исключения типа ClassFormatError.

Метод ответствен за сохранение ссылки на объект Class для загруженного класса в таблице загруженных классов, просматриваемой методом findLoadedClass.

Рассмотрим метод bytesForClass.

```
protected byte[]  bytesForClass(String name) throws
                                IOException, ClassNotFoundException{
  FileInputStream in = null;
  try {
    in = new FileInputStream(name + ".class");
    int length = in.available();   // число доступных байтов
    if (length == 0) throw new ClassNotFoundException(name);
    byte[ ] buf = new byte[length];
    in.read(buf);   // Считывание байтов
    return buf;
  }
  finally {
      if (in!=null) in.close();
  }
}
```

Таким образом полный код имеет вид:

```
import java.lang.reflect.*;
import java.io.*;
class MyClassLoader extends ClassLoader{
   public Class<?> findClass(String name) throws
                                                ClassNotFoundException {
      byte[ ] buf=ReadFromBuffer(name);
      if(name.equals("MyInterface1")){
            return findSystemClass(name);
      } else if(buf==null) {
            return findSystemClass(name);
      } else {
            return defineClass(name,buf,0,buf.length);
      }
   }
```

```
protected byte[] ReadFromBuffer(String name) throws
                                                ClassNotFoundException {
    FileInputStream in = null;
    try {
       in = new FileInputStream(name + ".class");
       int length = in.available(); // число доступных байтов
       if (length == 0) throw
                                  new ClassNotFoundException(name);
         byte[]  buf = new byte[length];
         in.read(buf); // Считывание байтов
         return buf;
     }
      catch(FileNotFoundException e){ return null;}
      catch(IOException e){ return null;}
      finally{
         try{ if (in!=null) in.close(); }
         catch(IOException e){ }
      }
}
```

```
protected synchronized Class
        loadClass(String name,boolean resolve) throws
                                        ClassNotFoundException{
            Class result= findClass(name);
            if (resolve) resolveClass(result);
             return result;
        }
}
```

```
public class Main1 {
   public static void main(String[] args) {
      try{
         String name="Myclass";
         ClassLoader  ld=new MyClassLoader();
         Class cl=Class.forName(name, true, ld);
         Constructor s=cl.getConstructor(int.class);
         MyInterface1
               ob=(MyInterface1)s.newInstance(
                                          new
  Integer(8));
         System.out.println(ob.func(3,5));
       }catch(Exception e){ };
     }
}
```

```
public interface MyInterface1{
    public int func(int a,int b);
}

public class Myclass implements MyInterface1 {
    private int a;
    public Myclass(int k) { a=k; }
    public int func(int a,int b){ return a+b; }
}
```