Конспект лекций по Java. Занятие 21

В.Фесюнов

1. Идентификация типа во время выполнения

Рассмотрим механизм идентификации типа во времени выполнения (`RTTI — run-time type identification`).

Данный механизм состоит из нескольких частей, которые отличаются своей важностью, сложностью и частотой использования.

- Базовые средства RTTI лежат в основе механизма полиморфизма. Соответственно, они используются постоянно при программировании на Java.
- При первом доступе к некоторому классу из программы, создается объект класса Class для данного класса. Это тоже относится к средствам RTTI.
- Механизмы определения типов (классов) объектов во время выполнения используются достаточно часто. Простейший из этих механизмов, операцию **instanceof** , мы уже рассматривали.
- Наиболее сложным механизмом RTTI является **рефлексия** . Это набор средств, позволяющих получить полную информацию о классе, такую как список полей, конструкторов, методов и т.д. Полученная информация может быть использована для создания объектов этого класса, вызова его методов.

1.1. Базовые средства RTTI

Любое приведение типов от базового к порожденному (нисходящее преобразование типов, **downcasting**) выполняется с динамическим контролем корректности типа.

Так, пусть у нас есть класс `Base` :

```
class Base { ... int f(); ... }
```

И от него порождены классы `Derived1` , `Derived2` `Derived3` . В программе мы

строим объекты этих классов и заносим их в список `ArrayList` . Далее мы хотим пройти по этим объектам и вызвать для каждого из них метод `f()` . Это будет выглядеть примерно так.

Где-то в программе у нас описан `ArrayList` .

```
ArrayList list;
```

В другой точке программы

```
... Iterator iter = list.iterator();
 while ( iter.hasNext() ) {
 ((Base)iter.next()).f();
 }
```

Здесь метод `next` интерфейса `Iterator` возвращает `Object` . Но мы знаем, что в списке находятся объекты классов `Derived1` , `Derived2` и `Derived3` , базовым классом для которых является `Base` . Мы не можем вызвать `f()` для `Object` . Поэтому мы приводим его к `Base` и вызываем `f()` .

Это шаблонный пример применения полиморфизма. Но самое главное, что нельзя забывать, это то, что в данном случае, как и в других подобных случаях, во время выполнения программы будет выполняться динамический контроль типов. Если в списке окажется объект какого-то другого класса, для которого `Base` не является базовым, то в процессе выполнения возникнет исключение `ClassCastException` .

Кроме того, динамические средства контроля типов в данной ситуации обеспечат вызов нужного метода `f()` . Мы ничего не говорили о классах `Derived1` , `Derived2` и `Derived3` . Но в любом из них мы можем переопределить (**override**) метод `f()` . При этом в данном фрагменте для каждого объекта из списка будет вызван метод `f()` его класса. И нам не нужно предпринимать для этого никаких специальных мер.

1.2. Объекты класса Class

Объекты класса `Class` создаются для каждого используемого в программе класса при первом доступе к нему. Первым доступом к классу может быть создание экземпляра класса или обращение к статическому методу или полю. Есть также и другие варианты в рамках механизма RTTI.

Имея доступ к объекту класса `Class` , соответствующему некоторому классу, можно создать экземпляр этого класса, получить имя класса и извлечь много различной другой информации о классе.

Обратимся к документации. Пакет `java.lang` класс `Class` .

Первое, на что следует обратить внимание, это отсутствие public конструкторов и описание самого класса:

```
public final class

Class
```

Это означает, что мы не можем ни унаследовать этот класс, ни явно создавать объекты этого класса. Класс `Class` устроен так, что нельзя создать более одного объекта для каждого класса. Виртуальная машина Java (VM) создает такой объект автоматически при первом обращении к классу.

Есть несколько способов получить этот объект для данного класса.

- При помощи статического метода класса `Class`

  ```
  public static Class forName(String className)
   throws ClassNotFoundException
  ```
- При помощи синтаксической конструкции, называемой *литералы объектов класса Class* .
- При помощи метода `getClass` класса `Object` .

В первом случае нам нужно знать полное имя класса. Во втором, класс должен быть доступен в данном java-файле (при помощи оператора `import`). Последний вариант может быть использован для получения объекта класса `Class` по объекту некоторого класса.

Можно применять любой из этих способов, в зависимости от того, что нам доступно в данной точке программы. Но при прочих равных условиях лучше отдать предпочтение второму из указанных способов — использованию *литералов объектов класса Class* .

1.2.1. Литералы объектов класса Class

Синтаксис данной конструкции весьма прост:

```
<Имя класса>.class
```

Например,

```
ArrayList.class
```

даст ссылку на объект класса `Class` для класса `java.util.ArrayList` .

Данный вариант предпочтительнее других потому, что он контролируется во время компиляции программы. Кроме того, он проще других вариантов.

Объекты класса `Class` создаются не только для всех классов, задействованных в программе, но и для примитивных типов. Причем, литералы объектов класса Class — один из двух способов получить доступ к этим объектам. Главное — знать, как это записать в программе.

Например, для типа `int` соответствующий литерал выглядит так: `int.class` .

Альтернативным способом является использование статического поля `TYPE` в классе-оболочке (*wrapper-class*)для данного простого типа. Например, для `int` : `Integer.TYPE` .

1.3. Определение типа объекта в программе

Иногда в программе необходимо определить или проверить тип объекта. Чаще всего такая необходимость возникает при проведении нисходящего преобразования классов (*downcasting*) для того, чтобы проверить допустимость такого преобразования.

Один из таких способов мы уже рассматривали на 8-м занятии. Это операция `instanceof` .

Другой способ является следствием рассмотренного выше материала по классу `Class` .

Третий способ состоит в использовании метода `isInstance` класса `Class` .

Рассмотрим и сравним эти способы.

На 8-м занятии мы рассматривали такой пример

```
Issue[] catalog = new Issue[] {
 new Journal("Play Boy"),
 new Newspaper("Спид-Инфо"),
 new Book("Война и мир",
"Л.Толстой"),
};
. . .
    for(int i = 0;
     i < catalog.length; i++) {
      if (catalog[i] instanceof Book )
       ((Book) catalog[i]).printAuthors(System.out);
           catalog[i].printName(System.out); }
```

В этом примере операция `instanceof` использована для выделения частного случая, когда печатное издание является книгой, и вывода на печать списка авторов книги.

Как видно из примера, левым операндом операции `instanceof` является объект, правым — класс.

Еще один способ сравнения состоит в сравнении объектов класса `Class` . Поскольку для каждого класса такой объект присутствует в единственном числе, то, получив две ссылки на объект класса `Class` , их можно просто сравнить на равенство. В приведенном примере это могло бы выглядеть так:

```
if ( catalog[i].getClass() == Book.class )
```

Однако этот способ сравнения отличается от сравнения с использованием `instanceof` . В частности, в данном примере он применим только в том случае, если класс `Book` не имеет наследников. Если же представить, что у класса `Book` есть наследники (например, `Fantasy` и `DetectiveStory`), то для объектов этих классов

проверка

```
if ( catalog[i].getClass() == Book.class )
```

не сработает и список авторов не отпечатается. Это связано с тем, что `catalog[i].getClass()` для объекта класса `Fantasy` или `DetectiveStory` выдаст объект `Class` , отличный от объекта `Book.class` .

В то же время проверка

```
if ( catalog[i] instanceof Book )
```

(т.е. с применением операции `instanceof`) отработает корректно, поскольку эта операция работает с учетом наследования классов.

Это отличие описанных выше способов сравнения нужно учитывать. Чаще всего нам требуется то, что делает `instanceof` , и лишь в редких случаях нас может интересовать точное сравнение классов, которое можно выполнить сравнивая ссылки на объекты класса `Class` .

И, наконец, есть вариант сравнения с использованием метода `isInstance` класса `Class` . По своей семантике этот способ ничем не отличается от применения операции `instanceof` . Их отличия чисто синтаксические. Так, для нашего примера применение `isInstance` выглядело бы так

```
if ( Book.class.isInstance(catalog[i]) )
```

1.4. Создание объектов с помощью метода newInstance

Вернемся к документации по классу `Class` . Среди методов данного класса есть метод `newInstance` .

```
public Object newInstance() throws InstantiationException, IllegalAccessException
```

Этот метод позволяет порождать объект того класса, для которого создан данный объект `Class` . Правда при этом применяется конструктор по умолчанию (*использование других конструкторов рассмотрим далее*).

Для знакомства с данным методом рассмотрим пример.

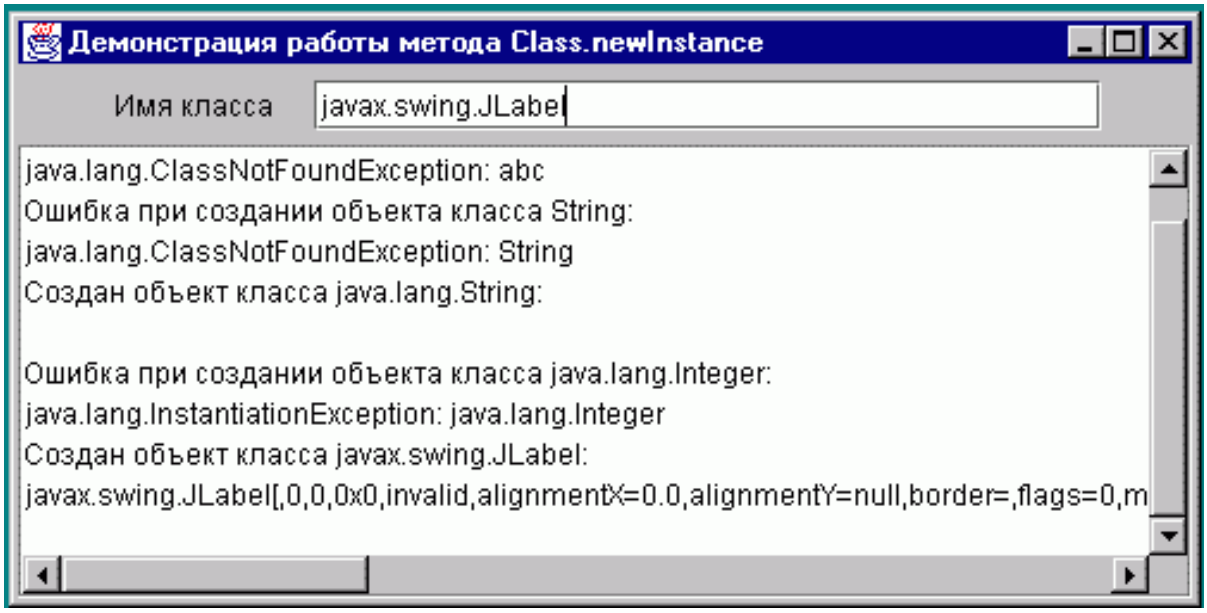
Файл **NewInstanceDemo.java** :

```
// NewInstanceDemo.java
// Демонстрация работы метода Class.newInstance

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class NewInstanceDemo extends JFrame {

    private JTextField fld = new JTextField(30);
    private JTextArea  msg = new JTextArea(8, 40);

    NewInstanceDemo() {
     super("Демонстрация работы метода Class.newInstance");

     try  {
       UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
     }
     catch(Exception e) {
     }

     setSize(500, 250);
     Container c = getContentPane();
     JPanel pn = new JPanel();
     c.add(pn, BorderLayout.NORTH);
     pn.add(new Label("Имя класса"));
     pn.add(fld);
     fld.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent evt) {
             createObject();
         }
     });
     JScrollPane pane = new JScrollPane(msg);
     c.add(pane, BorderLayout.CENTER);
     WindowListener wndCloser = new WindowAdapter() {
       public void windowClosing(WindowEvent e) {
         System.exit(0);
```

```
       }
     };
     addWindowListener(wndCloser);

     setVisible(true);
    }

    void createObject() {
        String className = fld.getText();
        try {
            Class cls = Class.forName(className);
            Object obj = cls.newInstance();
            msg.append("Создан объект класса "+className+":\n");
            msg.append(obj.toString()+"\n");
        } catch (Exception ex) {
            msg.append("Ошибка при создании объекта класса
"+className+":\n");
            msg.append(ex.toString()+"\n");
        }
    }

    public static void main(String args[]) {
        new NewInstanceDemo();
    }

}
```

Это диалоговая программа с текстовым полем для ввода имени класса и текстовой областью для вывода результатов создания объектов класса с использованием метода `newInstance`.

Оттранслируем и запустим программу. Сделаем несколько тестов. Ниже на рис. представлены результаты тестирования.



В качестве имен классов в данном тесте вводились 5 строк: *abc* , *String* , *java.lang.String* , *java.lang.Integer* и *javax.swing.JLabel* . Разберемся со случаями неудачных тестов.

- *abc* — нет такого класса.
- *String* — нужно задавать полное имя класса с указанием пакета.
- *java.lang.Integer* — для данного класса не существует конструктора по умолчанию.

Данная программа демонстрирует работу методов `forName` и `newInstance` . Основные десйствия программы сосредоточены в методе `createObject` . Этот метод сначала выбирает имя класса, введенное пользователем, из текстового поля. Потом, в блоке `try` , при помощи метода `forName` получает ссылку на объект `Class` для данного класса и создает объект этого класса методом `newInstance` .

Как метод `forName` , так и метод `newInstance` могут генерировать исключения. Пример тестирования, приведенный на рис., демонстрирует это. При вводе имен *abc* и *String* исключение `ClassNotFoundException` возникло в методе `forName` . При вводе *java.lang.Integer* возникло исключение `InstantiationException` в методе `newInstance` .

Мы рассмотрели базовые возможности RTTI. Нам осталось рассмотреть наиболее сложную часть RTTI — **рефлексию** .