Классы и интерфейсы

(продолжение)

instanceof

Оператор `instanceof` возвращает значение `true`, если объект является экземпляром данного класса, например:

```
Font obj = new Font("Courier", 1, 18);
if (obj instanceof java.awt.Font) {
   /*операторы*/
}
```

Внутренние (inner) классы

Нестатические вложенные классы принято называть внутренними (`inner`) классами.

Доступ к элементам внутреннего класса возможен из внешнего класса только через объект внутреннего класса, который должен быть создан в коде метода внешнего класса.

Объект внутреннего класса всегда скрыто хранит ссылку создавшего его объекта внешнего класса.

Внутренние (inner) классы

Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса, в то же время внешний класс может получить доступ к содержимому внутреннего класса только после создания объекта внутреннего класса.

Внутренние классы не могут содержать статические атрибуты и методы, кроме констант (`final static`).

Внутренние классы имеют право наследовать другие классы, реализовывать интерфейсы и выступать в роли объектов наследования.

Внутренние классы позволяют окончательно решить проблему множественного наследования, когда требуется наследовать свойства нескольких классов.

Внутренние (inner) классы

Пример (После компиляции объектный модуль, соответствующий внутреннему классу, получит имя Ship$Engine.class):

```
public class Ship {
    // поля и конструкторы
    // abstract, final, private, protected - допустимы
    public class Engine { // определение внутреннего класса
     // поля и методы
     public void launch() {
          System.out.println("Запуск двигателя");
     }
    }// конец объявления внутреннего класса

    public void init() {// метод внешнего класса
     // объявление объекта внутреннего класса
     Engine eng = new Engine();
     eng.launch();
    }
}
```

Внутренние (inner) классы

При таком объявлении объекта внутреннего класса `Engine` в методе внешнего класса `Ship` нет реального отличия от использования какого-либо другого внешнего класса, кроме объявления внутри класса `Ship`.

Использование объекта внутреннего класса вне своего внешнего класса возможно только при наличии доступа (видимости) и при объявлении ссылки в виде:

```
Ship.Engine obj = new Ship().new Engine();
```

Внутренние (inner) классы

Если внутренний класс наследуется обычным образом другим классом (после extends указывается ИмяВнешнегоКласса. ИмяВнутреннегоКласса), то он теряет доступ к полям своего внешнего класса, в котором он был объявлен.

```
public class Motor extends Ship.Engine {
 public Motor(Ship obj) {
     obj.super();
 }
}
```

В данном случае конструктор класса `Motor` должен быть объявлен с параметром типа `Ship`, что позволит получить доступ к ссылке на внутренний класс `Engine`, наследуемый классом `Motor`.

Вложенные (nested) классы

Если не существует необходимости в связи объекта внутреннего класса с объектом внешнего класса, то есть смысл сделать такой класс статическим.

Вложенный класс логически связан с классом-владельцем, но может быть использован независимо от него.

При объявлении такого внутреннего класса присутствует служебное слово `static`, и такой класс называется вложенным (`nested`).

Вложенные (nested) классы

Пример:

```
public class Ship {
   private int id;
   // abstract, final, private, protected - допустимы
   public static class LifeBoat {
    public static void down() {
       System.out.println("шлюпки на воду!");
    }
    public void swim() {
       System.out.println("отплытие шлюпки");
    }
   }
}
```

Вложенные (nested) классы

```
public class RunnerShip {
   public static void main(String[] args) {
   // вызов статического метода
    Ship.LifeBoat.down();
    // создание объекта статического класса
    Ship.LifeBoat lf = new Ship.LifeBoat();
    // вызов обычного метода
    lf.swim();
   }
}
```

Исключения

Исключения

Java предоставляет языковую конструкцию, которая носит название механизм исключений, и позволяет осуществлять обработку ошибок, возникающих в процессе выполнения программы.

Исключения возникают когда возникшая проблема не может быть решена в текущем контексте и невозможно продолжение работы программы.

Исключения

Исключение представляет собой событие, происходящее в процессе выполнения Java-программы в результате нормального хода выполнения команд.

При этом процесс обработки исключения называется перехватом исключения.

Процесс обработки ошибок в Java сводится к перехвату возбужденных исключений.

Исключения

Исключения представляют собой механизм взаимодействия между кодом, служащим для обнаружения ошибки и кодом, обрабатывающим ошибку.

Исключение возбуждается (`throws`) тогда, когда возникает определенная ситуация.

После возбуждения исключение перехватывается (`catch`).

Если исключение не перехватывается явным образом, то вызывается обработчик исключительных ситуаций, определенный в Java по умолчанию, который выводит информацию о стеке вызова текущего метода.

Исключения

Все исключения являются наследниками суперкласса `Throwable` и его подклассов `Error` и `Exception` из пакета `java.lang`.

Исключения делятся на две категории: проверяемые и непроверяемые.

К непроверяемым исключениям относятся исключения, унаследованные от классов `Error` и `RunTimeException`. Эти исключения возбуждаются ядром виртуальной машины Java.

Исключения

Проверяемые исключения описываются явно при определении метода или конструктора с помощью ключевого слова `throws`:

```
[модификаторы] тип метод (списокПараметров) throws списокИсключений
{
    телоМетода
}
    или

[модификаторы] конструктор (списокПараметров) throws списокИсключений
{
    телоКонструктора
}
```

Исключения

Для создания проверяемых исключений расширяется класс `Exception`, например:

```
class MyException extends Exception {...}
```

Исключения

Если при объявлении метода или конструктора класса указано обрабатываемое исключение, то программист при использовании этого метода или конструктора обязан исполнить одно из трех действий:

1. Перехватить и возбудить данное исключение при помощи ключевого слова `throws`.

2. Перехватить и обработать исключение при помощи управляющей конструкции `try-catch`.

3. В текущем методе или конструкторе не перехватывать исключение и при этом обязательно объявить текущий метод или конструктор с ключевым словом `throws`.

Исключения

Для перехвата проверяемых исключений, которые были возбуждены, используется следующая конструкция:

```
try
   {
   операторы, в работе которых может возбудиться исключение
   }
[catch типИсключения1 переменная1
   оператор1 ]
[catch типИсключения2 переменная2
   оператор2 ]
[...]]]
[finally
   {
   операторN
   }]
```

Исключения

Оператор после ключевого слова `try` исполняется до тех пор, пока он не будет успешно исполнен или до тех пор, пока не будет возбуждено исключение.

В случае возбуждения исключения будут просмотрены все условия `catch` для поиска подходящего исключения или одного из родительских классов этого исключения и исполнены соответствующие операторы после этих условий.

Исключения

Исполнение оператора после ключевого слова `finally` всегда происходит независимо от того, было ли возбуждено исключение или нет.

Это условие обычно используется для установки состояния объекта или для освобождения внешних ресурсов, например, для закрытия файла.

Исключения

Если исключение перехвачено определенным `catch` - выражением, то управление программой передается операторам этого `catch` - блока. Обычно в таких `catch` - блоках выполняются специальные действия, необходимые для устранения ошибок работы программы, вызванных возбужденным исключением.

По завершении работы операторов `catch` – блока управление все равно передается блоку операторов `finally`. Однако это не означает обязательность использования `finally` - блока в конструкции перехвата исключений.

Оператор throw

Исключительную ситуацию можно создать с помощью оператора `throw`, если объект-исключение уже существует, или инициализировать его прямо после этого оператора. Для этого может быть использован объект класса `Throwable` или объект его подкласса, а также ссылки на них.

```
throw объектThrowable;
```

Оператор throw

Объект-исключение может уже существовать или создаваться с помощью оператора `new`:

```
throw new IOException();
```

Инициализация объекта-исключения без оператора `throw` никакой исключительной ситуации не вызовет!

Отладочный механизм assertion

При помощи `assertion` можно сформулировать требования к входным, выходным и промежуточным данным методов классов в виде некоторых логических условий.

Инструкция assert:

```
assert (boolexp): expression;
assert (boolexp);
```

Отладочный механизм assertion

Пример:

```
int age = ob.getAge();
assert (age >= 0): "NEGATIVE AGE!!!";
// реализация
```

Массивы и строки

Массивы

Массивом в Java называется упорядоченный набор элементов. В качестве элементов массива могут выступать как данные примитивных типов, так и ссылки на объекты, включая ссылки на другие массивы.

Массив объявляется и создается при помощи выражения:

```
ТипЭлементов[] имяМассива =
    new типЭлементов [размерМассива]
```

или, эквивалентно:

```
ТипЭлементов имяМассива[] =
    new типЭлементов [размерМассива]
```

Массивы

Для создания массива необходимо выполнить следующие действия:

- Объявить массив - задать имя массива и тип элементов.
- Выделить для массива память - задать количество его элементов в вызове оператора new.
- Инициализировать массив, поместив в его элементы данные

Массивы являются объектами. Размер массива хранится в поле length объекта. Если массив имеет длину n, то корректными значениями индекса являются числа от 0 до n-1.

При обращении к массиву по некорректному индексу возбуждается исключение IndexOutOfBoundsException.

Массивы

Доступ к элементам массива имеет вид:

```
массив[индекс]
```

Доступ к полям массива имеет вид:

```
массив.поле
```

Доступ к полям и методам экземпляров класса, являющихся элементами массива, имеет вид:

```
массив[индекс].поле
массив[индекс].метод(параметрыМетода)
```

Массивы

Можно создавать массив, элементы которого являются массивами. Объявление и создание многомерных массивов имеет вид:

```
типЭлемента [][] ...имяМассива =
    new типЭлементов [размер1][размер2]...
```

Например:

```
int [][][] narr = new int [2][3][4];
```

При создании многомерного массива обязательно требуется указывать первый размер слева. Другие размеры для вложенных массивов можно указывать позже при помощи оператора new. Например:

```
int [][][] narr = new int [2][][];
...
narr[0] = new int [3][];
narr[1] = new int [3][];
```

Массивы

Инициализирующие значения массива задаются в фигурных скобках сразу же после объявления массива, например:

```
int [] nA ={1,2,3,4};
int [][] nB = {{1,0,0},{0,1,0},{0,0,1}};
String strArr[] ={"aaa", "bbb", "cde"+"xyz"};
int [][][] narr = {
   { {0}, {0, 1}, {0, 1, 2} },
   { {0, 1, 2}, {0, 1}, {0} }
};
```

Массивы - пример

```
public class ex1 {
    static int [][][] narr = {
       { {0}, {0, 1}, {0, 1, 2} },
    { {0, 1, 2}, {0, 1}, {0} }
    };
    public static void main( String [] args ) {
        System.out.println( narr.length );
        for ( int i = 0; i < narr.length; i++ ) {
            System.out.println( narr[i].length );
            for ( int j = 0; j < narr[i].length; j++ )
                System.out.print(
                                narr[i][j].length + " " );
            System.out.println();
        }
    }
}
```

Массивы - пример

Результат:

```
2
3
1 2 3
3
3 2 1
```

Массивы – класс java.util.Arrays

Класс Arrays содержит несколько удобных методов, предназначенных для работы с массивами.

```
static String toString(type[] a);
```

- возвращает строку с элементами a, заключенную в квадратные скобки и разделенную запятыми.

```
static type[] copyOf(type[] a, int length);
static type[] copyOf(type[] a, int start, int end);
```

-возвращает массив того же типа, что и a, длиной либо length, либо end - start, заполненный значениями из a.

Массивы – класс java.util.Arrays

```
static void sort(type[] a);
```

- сортирует массив, используя алгоритм быстрой сортировки

```
static int binarySearch(type[] a, type v);
static int binarySearch(type[] a, int start, int end, type
   v);
```

- использует алгоритм бинарного поиска для нахождения значения v; в случае успеха возвращается индекс найденного элемента; в противном случае возвращается отрицательное значение r; -r - 1 указывает на индекс позиции, куда должен быть вставлен искомый элемент, чтобы сохранился порядок сортировки.

Массивы – класс java.util.Arrays

```
static void fill(type[] a, type v);
```

- устанавливает все элементы массива в v.

```
static boolean equals(type[] a, type[] b);
```

- возвращает true, если массивы имеют равную длину и совпадают все их элементы в соответствующих позициях индекса.

Строки

Строки – это основной носитель текстовой информации. Строки не являются массивами символов типа char, это объекты соответствующего класса.

Пакет java.lang содержит классы String, StringBuilder и StringBuffer, поддерживающие работу со строками.

Эти классы объявлены как final, что лишает возможности создавать на их основе порожденные классы.

Строки – класс String

Каждая строка, создаваемая с помощью оператора new или с помощью литерала является объектом класса String.

Значение объекта String не может быть изменено после создания при помощи какого-либо метода класса. Любое изменение строки приводит к созданию нового объекта.

Если в выражении присутствует хотя бы один объект типа String, остальные объекты преобразуются в String с помощью метода toString().

Строки – класс String

Класс String поддерживает несколько конструкторов

```
String()
String(String str)
String(byte asciichar[])
String(char[] unicodechar)
String(StringBuffer sbuf)
String(StringBuilder sbuild)
```

и др.

Когда Java встречает литерал, заключенный в двойные кавычки, автоматически создается объект типа String, на который можно установить ссылку.

Строки – класс String

Пример:

```
public class ex2 {
    public static void main( String [] args ) {
        String str = "123“, strSave = str;
        str += "xyz";
        boolean flag = str == strSave;
        System.out.println(
             "The value of \"str == strSave\" is "
             + flag );
    }
}
```

Результат:

```
The value of "str == strSave" is false
```

Строки – класс String

String содержит следующие методы для работы со строками:

`String concat(String s); // или “+”`
– слияние строк;

`boolean equals(Object ob);`
`boolean equalsIgnoreCase(String s);`
- сравнение строк с учетом и без учета регистра соответственно

`int compareTo(String s);`
`int compareToIgnoreCase(String s);`
- лексикографическое сравнение строк с учетом и без учета регистра, путём вычитания кодов символов вызывающей и передаваемой в метод строк

boolean contentEquals(StringBuffer ob);
-сравнение строки и содержимого объекта типа StringBuffer;

Строки – класс String

```
String substring(int n);
```

- извлечение из строки подстроки, начиная с позиции n;

```
String substring(int n, int m);
```

- извлечение из строки подстроки длины m-n, начиная с позиции n. Нумерация символов в строке начинается с нуля;

```
int length();
```

- длина строки

Строки – класс String

```
boolean isEmpty();
```

- проверяет длину строки на 0

```
char charAt( int i );
```

- символ в позиции

```
String trim();
```

- удаление пробельных символов вначале и в конце строки

Строки – класс String

```
int indexOf( char c );

int indexOf( char c, int fromIdx );

int indexOf( String s );

int indexOf( String s, int fromIdx );
```

- поиск в прямом направлении символа или строки начиная с указанного индекса; возвращает индекс или -1

Строки – класс String

```
int lastIndexOf( char c );

int lastIndexOf( char c, int fromIdx );

int lastIndexOf( String s );

int lastIndexOf( String s, int fromIdx );
```

- поиск в обратном направлении символа или строки начиная с указанного индекса; возвращает индекс или -1

Строки – класс String

```
String toLowerCase();
String toUpperCase();
```

- изменение регистра символов

Строки – класс StringBuffer

Классы StringBuilder и StringBuffer являются “близнецами” и по своему предназначению близки к классу String, но, содержимое и размеры объектов классов StringBuilder и StringBuffer можно изменять.

Единственным отличием StringBuffer от StringBuilder является потокобезопасность StringBuffer. StringBuilder следует применять, если не существует вероятности использования объекта в конкурирующих потоках.

Объекты этих классов можно преобразовать в объект класса String методом toString() или с помощью конструктора класса String.

Строки – класс StringBuffer

Методы:

```
int length();
```
- размер строки

```
int capasity();
```
- размер буфера

```
void setLength( int n );
```
- изменение размера строки

```
void ensureCapacity( int n );
```
- изменение размера буфера

Строки – класс StringBuffer

```
char charAt( int i );
```

- символ в позиции

```
void setCharAt( int i, char ch  );
```

- изменение символа в позиции

```
StringBuffer reverse();
```

- обращение содержимого объекта

Строки – класс StringBuffer

```
StringBuffer append( String s );
StringBuffer append( int i );
StringBuffer append( Object obj );
```

- добавление аргумента в конец строки

```
StringBuffer insert( int idx, String s );
StringBuffer insert( int idx, char c );
StringBuffer insert( int idx, Object obj );
```

- добавление аргумента в указанную позицию строки

Строки – класс StringBuffer

```
StringBuffer delete(int idx, int idxLast);
```

- удаление подстроки

```
StringBuffer replace(int idx, int idxLast,
                     String str);
```

- замена подстроки

```
StringBuffer deleteCharAt(int pos);
```

- удаление символа

Строки – класс StringBuffer

```
String substring( int idx );
String substring( int idx, int idxLast );
```

- выделение подстроки

```
int indexOf( String s );
int indexOf( String s, int fromIdx );
```

- прямой поиск подстроки

```
int lastIndexOf( String s );
int lastIndexOf( String s, int fromIdx );
```

- обратный поиск подстроки

Строки – класс StringBuffer

При создании объекта StringBuffer конструктор резервирует некоторый объем памяти, что в дальнейшем позволяет быстро менять содержимое объекта, оставаясь в границах участка памяти, выделенного под объект. Размер резервируемой памяти при необходимости можно указывать в конструкторе.

Если длина строки StringBuffer после изменения превышает его размер, то ёмкость объекта автоматически увеличивается, оставляя при этом резерв для дальнейших изменений.

Если метод, вызываемый объектом StringBuffer, производит изменения в его содержимом, то это не приводит к созданию нового объекта, как в случае объекта String, а изменяет текущий объект StringBuffer.

Строки – класс StringBuffer

Пример:

```
public class DemoStringBuffer {
    public static void main(String[] args) {
     StringBuffer sb = new StringBuffer();
     sb.append("Java");
     System.out.println("строка ->" + sb);
     System.out.println("длина ->" + sb.length());
     System.out.println("размер ->" + sb.capacity());
     System.out.println("реверс ->" + sb.reverse());
    }
}
```

Результат:

```
строка ->Java
длина ->4
размер ->16
реверс ->avaJ
```

Строки – класс StringBuffer

Пример:

```
public class RefStringBuffer {
   public static void changeStr(StringBuffer s) {
    s.append(" Microsystems");
   }
   public static void main(String[] args) {
    StringBuffer str = new StringBuffer("Sun");
    changeStr(str);
    System.out.println(str);
   }
}
```

Результат:

```
Sun Microsystems
```

Строки – класс StringTokenizer

Используется для разбиения строки на лексемы.

Конструкторы:

```
public StringTokenizer(String string);
public StringTokenizer(String string,
                       String delimiters);
```

Строки – класс StringTokenizer

Методы:

```
public boolean hasMoreTokens();
public String nextToken();
public String nextToken(String newDelimiters);
```

Строки – класс StringTokenizer

Пример:

```
import java.util.StringTokenizer;
public class ex3{
    public static void main( String [] args ){
        String sentence = "It\'s a sentence, “+
            “it can be tokenized.";
        StringTokenizer st = new StringTokenizer
         (sentence, " ,.!?;-\n\r");
        while ( st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
}
```

Строки – класс StringTokenizer

Результат:

```
It's
a
sentence
it
can
be
tokenized
```

Интернационализация

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

Интернационализация

Характеристики интернационализированного приложения:

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

Локализация

Локализация - это процесс адаптации программного обеспечения для определенного региона или языка путем добавления специфических для данной местности компонентов и перевода текста.

Данные, зависящие от местности:

- текст;
- числа;
- денежные единицы;
- дата и время;
- изображения;
- цвета;
- звуки.

Локализация

Класс java.util.Locale позволяет учесть особенности региональных представлений алфавита, символов и проч. Автоматически виртуальная машина использует текущие региональные установки операционной системы, но при необходимости их можно изменять. Для некоторых стран региональные параметры устанавливаются с помощью констант, например: Locale.US, Locale.FRANCE. Для других стран объект Locale нужно создавать с помощью конструктора:

```
Locale myLocale = new Locale("bel", "BY");
```

Получить доступ к текущему варианту региональных параметров можно следующим образом:

```
Locale current = Locale.getDefault();
```

Локализация

Для создания приложений, поддерживающих несколько языков можно использовать возможности классов java.util.ResourceBundle и Locale.

Класс ResourceBundle предназначен для работы с текстовыми файлами свойств (расширение .properties).

Чтобы выбрать определенный объект ResourceBundle, следует вызвать метод ResourceBundle.getBundle(параметры). Следующий фрагмент выбирает text объекта ResourceBundle для объекта Locale, который соответствует английскому языку, стране Канаде и платформе UNIX.

```
Locale currentLocale = new Locale("en", "CA", "UNIX");
ResourceBundle rb =
    ResourceBundle.getBundle("text", currentLocale);
```

Локализация

Если объект ResourceBundle для заданного объекта Locale не существует, то метод getBundle() извлечет наиболее общий. В случае если общее определение файла ресурсов не задано, то метод getBundle() генерирует исключительную ситуацию MissingResourceException. Чтобы этого не произошло, необходимо обеспечить наличие базового файла ресурсов без суффиксов:

```
text.properties
```

В файлах свойств информация должна быть организована по принципу:

```
key1 = value1
key2 = value2
…
```

Локализация

Пример программы:

```
import java.util.*;
public class IntTest {
    static public void main(String args[]) {
     if (args.length != 2) {
          System.out.println("Format: java IntTest lang country");
          System.exit(-1);
     }
     String language = new String(args[0]);
     String country = new String(args[1]);
     Locale loc = new Locale(language, country);
     ResourceBundle messages =
          ResourceBundle.getBundle("MessagesBundle", loc);
          System.out.println(messages.getString("greeting"));
          System.out.println(messages.getString("inquiry"));
          System.out.println(messages.getString("farewell"));
    }
}
```

Локализация

Файлы ресурсов:

MessageBundle.properties:

```
greeting = Hello!
inquiry = How are you?
farewell = Goodbye!
```

MessageBundle_ru_RU.properties:

```
greeting = Привет!
inquiry = Как дела?
farewell = До свидания!
```