Лекция 5

ПЕРЕЧИСЛЕНИЯ, АВТОУПАКОВКА И АННОТАЦИИ

Перечисления

- *Перечисление* - это список именованных и логически связанных констант.
- Перечисления обычно используются для задания списка значений (состояния объекта, типы ошибок).
- Т.к. перечисления являются классами, то их возможности по сравнению с другими языками программирования значительно больше.
- Перечисления могут иметь конструкторы, методы и поля.

Перечисления

- Перечисление обозначается ключевым словом **enum**.

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

- Содержимое перечисления называется *константами перечисления*.
- Не смотря на то, что перечисление является классом, при его инициализации использовать оператор **new** не надо.

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

- См. annotations3

Методы values() и valueOf()

- Все перечисления по умолчанию содержат 2 предопределённых метода: **values()** и **valueOf()**.

```
public static enum-type [] values()
public static enum-type
valueOf(String str)
```

- Метод **values()** возвращает массив, который содержит список констант перечисления.
- **ValueOf()** возвращает значение перечисления, совпадающее с именем параметра.
- См. Annotations4.

Перечисление – это класс

- Перечисление может иметь конструкторы, поля и методы, и даже реализовывать интерфейсы.
- При определении конструктора для **enum**, он вызывается каждый раз при создании константы перечисления.
- Перечисление не может содержать в себе другой класс.
- Перечисление не может быть суперклассом.
- См. Annotations5.

Перечисления наследуются от Enum

- При объявлении перечисления происходит автоматическое наследование от класса **java.lang.Enum**.
- С помощью метода **ordinal()** можно получить значение порядкового номера, которое имеет константа перечисления.

```
for(Apple a: Apple.values())
    System.out.println(a + " " + a.ordinal());
```

- Для сравнения порядковых номеров двух констант одного перечисления используется метод **compareTo()**.

```
If (ap.compareTo(ap2) < 0)
    System.out.println(ap + "comes before " + ap2);
```

Пример перечисления

- **Смотри программу Annotation7.**

Класс wrapper (оболочка)

- Примитивные типы (int, double) более производительны в вычислениях, чем ссылочные.
- Но нельзя использовать примитивный тип в качестве ссылки на метод.
- В таких случаях Java использует *классы-оболочки*, которые являются классами, содержащими примитивные типы внутри объекта.
- Оболочками являются **Double**, **Float**, **Long**, **Integer**, **Short**, **Byte**, **Character**, and **Boolean**.

Character

- **Character** - это оболочка вокруг типа **char**.

```
Character(char ch)
```

- Начиная с JDK 9, **Character** является устаревшим.
- Рекомендуется вместо него использовать статический метод **valueOf()**.

```
static Character valueOf(char ch)
```

- Для получения значения экземпляра объекта Character используют метод charValue().

```
char charValue()
```

Boolean

- **Boolean** – это оболочка вокруг **boolean**-значений.

```
Boolean(boolean boolValue)
Boolean(String boolString)
```

- Начиная с JDK 9, **Boolean** является устаревшим.
- Рекомендуется вместо него использовать статический метод **valueOf()**.

```
static Boolean valueOf(boolean boolValue)
static Boolean valueOf(String boolString)
```

- Для получения значения экземпляра объекта **Boolean** используют метод **booleanValue()**:

```
boolean booleanValue()
```

Оболочка числовых типов

- Все оболочки числовых типов наследуются от абстрактного класса **Number**, который определяет методы, возвращающие значение объекта в каждом из числовых форматов.

```
byte byteValue()
double doubleValue()
float floatValue()
int intValue()
long longValue()
short shortValue()
```

- Для получения экземпляра объекта оболочки рекомендуется использовать один из методов **valueOf()**.

```
static Integer valueOf(int val)
static Integer valueOf(String valStr) throws
NumberFormatException
```

Автоупаковка

- Автоупаковка - это процесс, при котором примитивный тип автоматически инкапсулируется (упаковывается) в эквивалентный тип-оболочку.
- Необходимость создавать экземпляр объекта явно отпадает.
- Авторараспаковка - это процесс, который значение запакованное в экземпляре объекта автоматически извлекает (распаковывает) из типа-оболочки.

```
Integer iOb = 100; // автоупаковка типа int
int i = iOb; // авторараспаковка
```

Автоупаковка и методы

- Автоупаковка автоматически происходит, когда примитивный тип необходимо конвертировать в экземпляр объекта.
- Автораспаковка необходима, когда объект необходимо конвертировать в примитивный тип.

```
class Autobox {
static int m(Integer v) {
  return v;
}

public static void main (String args[ ]) {
Integer iOb = m(100);
}
```

Автоупаковка/распаковка в расширениях

- Внутри расширения числовые объекты автоматически распаковываются.

```
Integer iOb;
int i;
iOb = 100;
++iOb; // происходит распаковка, вычисление
       // и упаковка
```

Автоупаковка/распаковка значений Boolean и Character

```
Boolean b = true;
if (b) System.out.println(“b is true”);
Character ch = ‘x’;
char ch2 = ch;
```

Автоупаковка/распаковка, помогает не допускать ошибки

```
// Ошибка при ручной распаковке
public class Main {

    public static void main(String[] args) {

   // Автоупаковка значения 1000
        Integer iOb = 1000;

    // Ручная распаковка в байт!
        int i = iOb.byteValue();

    // Не покажет 1000
        System.out.println(i);
    }
}
```

Предупреждение!

- Из-за автоупаковки и автораспаковки возникает желание использовать исключительно типы **Integer** или **Double**, совершенно не используя примитивные типы.

```
Double a, b, c;
a = 10.0;
b = 4.0;
c = Math.sqrt(a*a + b*b);
System.out.println("Hypotenuse is " + c);
```

- Данный код будет работать, но медленнее, чем с примитивными типами.

Аннотации

- Аннотация – специальная форма синтаксических метаданных, которая добавляется в файлы исходного кода и не изменяет выполнение программы.
- Информация из аннотации может использоваться различными инструментальными средствами как при разработке, так и при развёртывании ПО.
- Например, аннотации могут обрабатываться генератором исходного кода.
- Термин *метадата* обозначает то же самое, но *аннотация* более объемлющий и распространённый.

Аннотации

- Аннотация создаётся через механизм, основанный на интерфейсе.

```
@interface MyAnno {
   String str();
   int val();
}
```

- Методы аннотаций действуют, как поля.

```
@MyAnno(str = "Annotation Example", val =
100)
public static void myMeth() { // ...
```

- Аннотация не может наследоваться от класса.

Политики удержания

- Java определяет 3 политики удержания, которые находятся внутри перечисления **java.lang.annotation.RetentionPolicy**: **SOURCE**, **CLASS** и **RUNTIME**.
- Аннотация с политикой удержания **SOURCE** удерживается только в файле с исходным кодом и отбрасывается в ходе компиляции.
- Аннотация с политикой удержания **CLASS** храниться файле **.class** в ходе компиляции.
- Аннотация с политикой удержания **RUNTIME** храниться в файле **.class** в ходе компиляции доступна через JVM в ходе выполнения.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}
```

Получение аннотаций во время выполнения с использованием рефлексии

- Рефлексия - это механизм получения информации о классе во время выполнения программы.
- Одним из самых простых способов рефлексии класса является метод **getClass()**, определённый в **Object**.

```
class Meta {
  @MyAnno(str = “Ann. example”, val = 100)
  public static void myMeth() {
    Meta ob = new Meta();
    try {
      Class<?> c = ob.getClass();
      Method m = c.getMethod(“myMeth”);
```

…

Пример рефлексии

- Смотри программу Annotation22.

Получение всех аннотаций

- Для получения всех аннотаций, имеющих политику удержания **RUNTIME,** необходимо вызвать метод **getAnnotations()**:

```
Annotation[] getAnnotations()
```

- Пример:

```
try {
  Annotation annots[] =
ob.getClass().getAnnotations();
  …
  Method m =
ob.getClass().getMethod("myMeth");
  annots = m.getAnnotations();
```

Интерфейс AnnotatedElement

- Данный интерфейс поддерживает рефлексию для аннотаций и реализовывается в классах **Method**, **Field**, **Constructor**, **Class**, и **Package**.
- Метод **getDeclaredAnnotations()** возвращает массив всех не наследуемых аннотаций, представленных в вызываемом экземпляре класса.
- Метод **isAnnotationPresent()** возвращает **true** если аннотация описана в вызываемом экземпляре объекта.

Использование значений по умолчанию

- Значение по умолчанию задаётся добавлением оператора **default** к объявлению поля или метода класса:

```
type member() default value;
```

- Например:

```
@interface MyAnno {
  String str() default “Testing”;
  int val() default 9000;
}
```

Маркер-аннотации

- Маркер-аннотация - это специальный вид аннотаций, который ничего не содержит.
- Данный вид аннотаций служит для пометки части кода.
- Например:

```
@MyMarker
public static void myMeth() {…}
```

Аннотации с одним элементом

- Они работают как обычные аннотации за исключением того, что позволяют в краткой форме получить значение элемента.
- В таких аннотациях не надо указывать имя присваиваемого значения.
- Например:

```
@interface Mysingle {
   int value();
}
…
@MySingle(100)
```

Встроенные аннотации

- Из пакета **java.lang.annotation**: **@Retention**, **@Documented** (это аннотация, подлежащая документированию), **@Target** (указывает, для каких элементов применяется аннотация) и **@Inherited** (используется для указания суперклассов).
- **Из пакета java.lang**: **@Override**, **@Deprecated**, **@FunctionalInterface** (это функциональный интерфейс, имеющий единственный абстрактный метод), **@SafeVarargs** (указывает на безопасность совершаемых действий) и **@SuppressWarnings** (подавляет вывод предупреждений).

Аннотации типов

- Аннотации могут применяться и к используемому типу.
- Аннотации типов указывают на необходимость проведения дополнительных проверок кода.
- Аннотация типа должна включать цель **ElementType.TYPE_USE**.

```
void myMeth() throws @TypeAnno NullPointerException
{ // ...
int myMeth(@TypeAnno SomeClass this, int i, int j) {
// ...
```

Повторяющиеся аннотации

- Начиная с JDK 8, у аннотаций появилась возможность повторного использования на одном элементе.
- Для этого нужно использовать аннотацию **@Repeatable** из пакета **java.lang.annotation**.

```
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(MyRepeatedAnnos.class)
@interface MyAnno {
  String str() default “Testing”;
  int val() default 9000;
}

@Retention(RetentionPolicy.RUNTIME)
@interface MyRepeatedAnnos {
  MyAnno[] value();
}

Class RepeatAnno {
@MyAnno(str = “First annotation”, val = -1)
@MyAnno(str = “Second annotation”, val = 100)
public static void myMeth(String str, int i)
…
```