Лекция 4 Интерфейсы, перечисления и классы обертки

План

1. Отношения между классами
2. Приведение ссылочных типов
3. Перечисления
4. Оболочки типов
5. Автоупаковка
6. Перегрузка с дополнительными факторами

Отношения между классами

Большая часть классов приложения связаны между собой. В этом разделе рассмотрим какие бывают отношения между классами в Java.

Отношения между классами

IS-A отношения

В ООП принцип IS-A основан на наследовании классов или реализации интерфейсов. Например, если класс HeavyBox наследует Box, мы говорим, что HeavyBox является Box (HeavyBox IS-A Box). Или другой пример - класс Lorry расширяет класс Car. В этом случае Lorry IS-A Car.

Тоже самое относится и к реализации интерфейсов. Если класс Transport реализует интерфейс Moveable, то они находятся в отношении Transport IS-A Moveable.

Отношения между классами

HAS-A отношения

HAS-A отношения основаны на использовании. Выделяют три варианта отношения HAS-A: ассоциация, агрегация и композиция.

Отношения между классами

Начнем с ассоциации. В этих отношениях объекты двух классов могут ссылаться друг на друга. Например, класс Horse HAS-A Halter если код в классе Horse содержит ссылку на экземпляр класса Halter:

```
public class Halter {}
```

```
public class Horse{
    private Halter halter;
}
```

Отношения между классами

Агрегация и композиция являются частными случаями ассоциации. Агрегация - отношение когда один объект является частью другого. А композиция - еще более тесная связь, когда объект не только является частью другого объекта, но и вообще не может принадлежат другому объекту. Разница будет понятна при рассмотрении реализации этих отношений.

Отношения между классами

Агрегация

Объект класса Halter создается извне Horse и передается в конструктор для установления связи. Если объект класса Horse будет удален, объект класса Halter может и дальше использоваться, если, конечно, на него останется ссылка:

```
public class Horse {
    private Halter halter;

    public Horse(Halter halter) {
        this.halter = halter;
    }
}
```

Отношения между классами

Композиция

Теперь посмотрим на реализацию композиции. Объект класса Halter создается в конструкторе, что означает более тесную связь между объектами. Объект класса Halter не может существовать без создавшего его объекта Horse:

```
public class Horse {
    private Halter halter;

    public Horse() {
        this.halter = new Halter();
    }
}
```

Приведение ссылочных типов

В предыдущих уроках мы рассматривали преобразование примитивных типов. В этом разделе рассмотрим какие существуют варианты приведения ссылочных типов в языке Java. В общем выделяют два варианта - это сужение и расширение.

Приведение ссылочных типов

Расширение типов в Java

Первый вариант приведения - это расширение. Расширение означает переход от более конкретного типа к менее конкретному, то есть переход от детей к родителям.

Приведение ссылочных типов

Подобно случаю с примитивными типами, этот переход производится самой JVM при необходимости и незаметен для разработчика. Если класс Box суперкласс, а HeavyBox - его наследник, то объект типа HeavyBox можно неявно преобразовать к Box:

```
Box heavyBox = new HeavyBox(15, 10, 20, 5);
```

Также расширяющим являются преобразованием от null-типа к любому объектному типу:

```
Box box = null;
```

Приведение ссылочных типов

Сужение типов в Java

Обратный переход, то есть движение по дереву наследования вниз, к наследникам, является сужением. Объект box типа Box приводим к HeavyBox:

```
Box box = new HeavyBox();
HeavyBox heavyBox = (HeavyBox) box;
```

Приведение ссылочных типов

Следующий пример показывает зачем нужны сужающие преобразования.

Допустим переменная box1 типа Box6 указывает на объект типа HeavyBox1 (Box6 - это суперкласс, HeavyBox1 - его наследник). Мы хотим вывести на консоль значение переменной класса weight для объекта box1. Но переменная weight объявлена в классе HeavyBox1, поэтому доступа к ней через box1 мы не имеем. Для того чтобы обратиться к весу, нам необходимо сделать приведение к HeavyBox1: HeavyBox1 heavyBox1 = (HeavyBox1) box1.

Приведение ссылочных типов

При попытке приведения переменной box2, указывающей на объект типа ColorBox, к HeavyBox1, возникнет ошибка ClassCastException времени выполнения. ColorBox тоже является наследником класса Box6, поэтому ошибки компиляции не возникнет.

```
public class CastingDemo1 {
    public static void main(String[] args) {
        Box6 box1 = new HeavyBox1();
        // System.out.println(box1.weight);

        HeavyBox1 heavyBox1 = (HeavyBox1) box1;
        System.out.println("Вес: " + heavyBox1.weight);

        Box6 box2 = new ColorBox();
        HeavyBox1 heavyBox2 = (HeavyBox1) box2; //ClassCastException

        Box6 box3 = new Box6();
        HeavyBox1 heavyBox3 = (HeavyBox1) box3; //ClassCastException
    }
}
```

Приведение ссылочных типов

instanceof keyword - это двоичный оператор, используемый для проверки, является ли объект (экземпляр) подтипом данного типа.

```
interface Domestic {}
class Animal {}
class Dog extends Animal implements Domestic {}
class Cat extends Animal implements Domestic {}
```

Приведение ссылочных типов

Представьте объект dog, созданный с помощью Object dog = new Dog(), а затем:

```
dog instanceof Domestic // true - Dog implements Domestic
dog instanceof Animal   // true - Dog extends Animal
dog instanceof Dog      // true - Dog is Dog
dog instanceof Object   // true - Object is the parent type of all objects
```

Приведение ссылочных типов

Однако, с Object animal = new Animal();

```
animal instanceof Dog // false
```

потому что Animal является супертипом dog и, возможно, менее "уточненным".

```
dog instanceof Cat
```

Будет false. Это связано с тем, что dog не является подтипом и не реализует его.

Приведение ссылочных типов

Несовместимые преобразования в Java

Преобразования возможны только внутри одной иерархии.

Данный пример не скомпилируется:

```
Box6 box1 = new HeavyBox1();
String str = (String) box1;
```

Приведение ссылочных типов

Переходы между массивами и примитивными типами являются запрещенными:

```
public class ArrayCastingDemo1 {
    public static void main(String[] args) {
        int[] array = new int[5];
        //int someNumber =array;
        int someNumber = array[0];
    }
}
```

Приведение ссылочных типов

Массив, основанный на примитивном типе, принципиально нельзя преобразовать к типу массива, основанному на ссылочном типе, и наоборот:

```
public class ArrayCastingDemo2 {
    public static void main(String[] args) {
        Integer[] array1 = new Integer[4];
        int[] array2 = new int[4];
        //  array1 = array2;
        //  array2 = array1;
    }
}
```

Приведение ссылочных типов

Преобразования между типами массивов, основанных на различных примитивных типах, невозможно:

```
public class ArrayCastingDemo3 {
    public static void main(String[] args) {
        int[] array1 = new int[5];
        long[] array2 = new long[5];
        // array2 = array1;
    }
}
```

Приведение ссылочных типов

Массив, основанный на типе HeavyBox, можно привести к массиву, основанному на типе Box, если сам тип HeavyBox приводится к типу Box.

```
public class ArrayCastingDemo5 {
    public static void main(String[] args) {
        rightConversion();
        wrongConversion();
    }

    private static void rightConversion() {
        Box6[] boxArray = new Box6[5];
        HeavyBox1[] heavyBoxArray = new HeavyBox1[6];
        boxArray = heavyBoxArray;
    }

    private static void wrongConversion() {
        Box6[] boxArray = new Box6[5];
        HeavyBox1[] heavyBoxArray = new HeavyBox1[6];
        //ошибка времени выполнения
        heavyBoxArray = (HeavyBox1[]) boxArray;
    }
}
```

Перечисления

В Java 5 были введены перечисления, которые создаются с использованием ключевого слова enum. Перечисления указывают возможные значения для какого-то явления. Например, вы открыли кофейню, в которой продаются три возможные варианты кофе - BIG, HUGE и OVERWHELMING. Других вариантов быть не может. Если задавать значения с помощью String, можно выбрать любое другое значение, например - MIDDLE, SMALL. Задавая перечисления, вы ограничиваете возможные варианты:

```
public enum CoffeeSize {
    BIG, HUGE, OVERWHELMING
}
```

Перечисления

В простейшей форме перечисления - это список именованных констант. Каждая константа перечисления (BIG, HUGE и OVERWHELMING) является объектом класса, в котором она определена. Константы перечисления являются static final и не могут быть изменены после создания.

Перечисления можно представить в виде класса, содержащего константы, например:

```
public class CoffeeSize5 {
    public static final String BIG = "BIG";
    public static final String HUGE = "HUGE";
    public static final String OVERWHELMING = "OVERWHELMING";
}
```

Перечисления

Но у перечислений гораздо больше преимуществ по сравнению с таким классом. Какие - рассмотрим чуть позже.

Можно создавать переменные типа перечисления. При этом не используется оператор new. Переменная перечисления объявляется и применяется практически так же, как и переменные примитивных типов:

```
public class CoffeeSizeDemo1 {
    public static void main(String[] args) {
        CoffeeSize coffeeSize = CoffeeSize.BIG;
        if (coffeeSize == CoffeeSize.BIG) {
            System.out.println(coffeeSize);
        }
    }
}
```

Перечисления

Значения перечислимого типа можно также использовать в управляющем операторе switch. В выражениях ветвей case должны использоваться константы из того же самого перечисления, что и в самом операторе switch. В выражениях ветвей case имена констант указываются без уточнения имени их перечислимого типа. Тип перечисления в операторе switch уже неявно задает тип enum для операторов case.

```
public class CoffeeSizeDemo2 {
    public static void main(String[] args) {
        CoffeeSize coffeeSize = CoffeeSize.BIG;
        switch (coffeeSize) {
            case BIG:
                System.out.println("Дайте мне большую чашку кофе!");
                break;
            case HUGE:
                System.out.println("Дайте мне огромную чашку кофе!");
                break;
            case OVERWHELMING:
                System.out.println("Дайте мне громадную чашку кофе!");
                break;
            default:
                System.out.println("Чашка не выбрана.");
        }
    }
}
```

Перечисления

Перечисление в Java относится к типу класса, но перечисление НЕ может наследоваться от другого класса и НЕ может быть суперклассом.

Все перечисления автоматически наследуют от класса java.lang.Enum. В этом классе определяется ряд методов, доступных для использования во всех перечислениях: ordinal(), compareTo(), equals(), values() и valueOf().

Перечисления также неявно наследуют интерфейсы Serializable и Comparable.

Рассмотрим методы класса java.lang.Enum.

Перечисления

Метод values() возвращает массив, содержащий список констант перечислимого типа:

```
public class CoffeeSizeValuesDemo {
    public static void main(String[] args) {
        System.out.println("Константы перечислимого типа CoffeeSize:");
        CoffeeSize[] coffeeSizes = CoffeeSize.values();
        for (CoffeeSize c : coffeeSizes) {
            System.out.println(c);
        }
    }
}
```

Перечисления

Результат выполнения:

```
Константы перечислимого типа CoffeeSize:
BIG
HUGE
OVERWHELMING
```

Перечисления

Статический метод valueOf() возвращает константу перечислимого типа, значение которой соответствует символьной строке, переданной в качестве аргумента. Можно сказать, что этот метод преобразует значение String в перечисление:

```
public class CoffeeSizeDemo3 {
    public static void main(String[] args) {
        CoffeeSize coffeeSize = CoffeeSize.valueOf("BIG");
        System.out.println("Переменная coffeeSize содержит " + coffeeSize);
    }
}
```

Перечисления

Вызвав метод ordinal(), можно получить значение, которое обозначает позицию константы в списке констант перечислимого типа. Порядковые значения начинаются с нуля:

```
public class CoffeeSizeDemo4 {
    public static void main(String[] args) {
        for (CoffeeSize c : CoffeeSize.values()) {
            System.out.println(c + " " + c.ordinal());
        }
    }
}
```

```
BIG 0
HUGE 1
OVERWHELMING 2
```

Перечисления

С помощью метода int compareTo(типПеречисления e) можно сравнить порядковые значения двух констант одного и того же перечислимого типа. Метод возвращает значение типа int.

Если порядковое значение вызывающей константы меньше, чем у константы e (this < e), то метод compareTo() возвращает отрицательное значение.

Если порядковые значения обеих констант одинаковы (this == e), возвращается нуль.

Если порядковое значение вызывающей константы больше, чем у константы e (this > e), то возвращается положительное значение.

```
public class CoffeeSizeCompareTo {
    public static void main(String[] args) {
        CoffeeSize bigCup = CoffeeSize.BIG;
        CoffeeSize hugeCup = CoffeeSize.HUGE;
        CoffeeSize anotherBigCup = CoffeeSize.BIG;
        CoffeeSize overwhelmingCup = CoffeeSize.OVERWHELMING;

        System.out.println("bigCup.compareTo(hugeCup): " + bigCup.compareTo
            (hugeCup));
        System.out.println("hugeCup.compareTo(bigCup): " + hugeCup.compareTo
            (bigCup));
        System.out.println("bigCup.compareTo(anotherBigCup): " + bigCup
            .compareTo(anotherBigCup));
        System.out.println("bigCup.compareTo(overwhelmingCup): " + bigCup
            .compareTo(overwhelmingCup));
    }
}

enum CoffeeSize {
    BIG, HUGE, OVERWHELMING
}
```

Перечисления

Результат:

```
bigCup.compareTo(hugeCup): -1
hugeCup.compareTo(bigCup): 1
bigCup.compareTo(anotherBigCup): 0
bigCup.compareTo(overwhelmingCup): -2
```

Перечисления

Вызвав метод equals(), переопределяющий аналогичный метод из класса Object, можно сравнить на равенство константу перечисления с любым другим объектом. Но оба эти объекта будут равны только в том случае, если они ссылаются на одну и ту же константу из одного и того же перечисления. Простое совпадение порядковых значений не вынудит метод equals() возвратить логическое значение true, если две константы относятся к разным перечислениям.

Перечисления

При сравнении констант перечислений, можно использовать оператор "==" - он будет работать также, как и метод equals().

```
public class CoffeeSizeEquals {
    public static void main(String[] args) {
        CoffeeSize bigCup = CoffeeSize.BIG;
        CoffeeSize hugeCup = CoffeeSize.HUGE;
        CoffeeSize anotherBigCup = CoffeeSize.BIG;

        System.out.println("bigCup.equals(hugeCup): " + bigCup.equals(hugeCup));
        System.out.println("bigCup.equals(anotherBigCup): " + bigCup.equals(anotherBigCup));
        System.out.println("bigCup == anotherBigCup: " + (bigCup == anotherBigCup));
    }
}
```

```
bigCup.equals(hugeCup): false
bigCup.equals(anotherBigCup): true
bigCup == anotherBigCup: true
```

Перечисления

Создать экземпляр перечисления с помощью оператора new нельзя, но в остальном перечисление обладает всеми возможностями, которые имеются у других классов. А именно - в перечисления можно добавлять конструкторы, переменные и методы. Конструкторы перечисления являются private по умолчанию.

Допустим мы хотим задать размер нашей чашки кофе в миллилитрах. Для этого введем переменную ml в перечисление и геттер метод getMl(). Добавим конструктор, на вход которого будем задавать значение для миллилитров.

Обратите внимание, что при объявлении конструктора не указан модификатор доступа - он private по умолчанию. Уже говорилось, что нельзя создавать объекты перечисления используя оператор new. Как же тогда вызвать наш конструктор? Для вызова конструктора перечисления после указания константы ставятся круглые скобки, в которых передается нужное значение.

Перечисления

```
public enum CoffeeSize2 {
    // 100, 150 и 200 передаются в конструктор
    BIG(100), HUGE(150), OVERWHELMING(200);

    private int ml;

    CoffeeSize2(int ml) {
        this.ml = ml;
    }

    public int getMl() {
        return ml;
    }
}
```

Перечисления

Методы для перечисления вызываются так же, как и для обычного объекта. В следующем классе мы перебираем все константы нашего перечисления и для каждого вызываем метод getMl():

```
public class CoffeeSizeMlDemo {
    public static void main(String[] args) {
        for (CoffeeSize2 coffeeSize : CoffeeSize2.values()) {
            System.out.println(coffeeSize + " " + coffeeSize.getMl());
        }
    }
}
```

Перечисления

Конструкторы в перечислении могут быть перегружены, как показано в следующем примере. Для вызова конструктора без параметров просто не пишите скобки после константы:

```
public enum CoffeeSize3 {
    BIG(100), HUGE, OVERWHELMING(200);

    private int ml;

    CoffeeSize3(int ml) {
        this.ml = ml;
    }

    CoffeeSize3() {
        this.ml = -1;
    }

    public int getMl() {
        return ml;
    }
}
```

Перечисления

Перечисления могут быть объявлены: отдельным классом или как член класса. Но НЕ могут быть объявлены внутри метода.

В этом пример перечисление CoffeeSize объявлено внутри класса Coffee3:

```
public class Coffee3 {
    enum CoffeeSize { BIG, HUGE, OVERWHELMING }

    CoffeeSize size;
}
```

Перечисления

Для обращения к такому перечислению необходимо использовать имя внешнего класс:

```
public class CoffeeTest2 {
    public static void main(String[] args) {
        Coffee3 drink = new Coffee3();
        drink.size = Coffee3.CoffeeSize.BIG; // имя внешнего класса необходимо
    }
}
```

```
public class CoffeeTest3 {
    public static void main(String[] args) {
        // Неправильно! Нельзя объявлять перечисления внутри метода!
        /*enum CoffeeSize {BIG, HUGE, OVERWHELMING}
        Coffee drink = new Coffee();
        drink.size = CoffeeSize.BIG;*/
    }
}
```

Перечисления

Перечисления могут быть объявлены: отдельным классом или как член класса. Но НЕ могут быть объявлены внутри метода.

В этом пример перечисление CoffeeSize объявлено внутри класса Coffee3:

```
public class Coffee3 {
    enum CoffeeSize { BIG, HUGE, OVERWHELMING }

    CoffeeSize size;
}
```

Перечисления

Для перечислений можно переопределять методы, но это не совсем обычное переопределение.

Добавим в наше перечисление метод getLid(), который возвращает код крышки для чашки кофе. Для всех констант подходит код B, который возвращает этот метод, кроме константы OVERWHELMING. Для OVERWHELMING чашки нужен код A. Переопределим метод getLid() для этой константы. Как это делается? После объявления константы открываем фигурные скобки, в которых и переопределяем этот метод. Если необходимо переопределить несколько методов, это делается в этих же фигурных скобках.

Перечисления

```
public enum CoffeeSize4 {
    BIG(100),
    HUGE(150),
    OVERWHELMING(200) {
        @Override
        public String getLidCode() {
            return "A";
        }
    };

    private int ml;

    CoffeeSize4(int ml) {
        this.ml = ml;
    }

    public int getMl() {
        return ml;
    }

    public String getLidCode() {
        return "B";
    }
}
```

```
public class CoffeeSizeDemo7 {
    public static void main(String[] args) {
        for (CoffeeSize4 coffeeSize : CoffeeSize4.values()) {
            System.out.println(coffeeSize + " " + coffeeSize.getLidCode());
        }
    }
}
```

```
BIG B
HUGE B
OVERWHELMING A
```

Перечисления

Перечисления не могут наследовать другие классы, но могут реализовывать интерфейсы. Например, следующее перечисление реализует интерфейс Runnable:

```
public enum Currency implements Runnable {
    PENNY(1), NICKLE(5), DIME(10), QUARTER(25);
    private int value;

    Currency(int value) {
        this.value = value;
    }

    @Override
    public void run() {
        System.out.println("Перечисления в Java могут реализовывать интерфейсы");
    }
}
```

Оболочки типов

Очень часто необходимо создать класс, основное назначение которого содержать в себе какое-то примитивное значение. Например, как мы увидим в следующих занятиях, обобщенные классы и в частности коллекции работают только с объектами. Поэтому, чтобы каждый разработчик не изобретал велосипед, в Java SE уже добавлены такие классы, которые называются оболочки типов (или классы обертки, Wrappers).

Оболочки типов

К оболочкам типов относятся классы Double, Float, Long, Integer, Short, Byte, Character, Boolean, Void. Для каждого примитивного значения и ключевого слова void есть свой класс-двойник. Имя класса, как вы видите, совпадает с именем примитивного значения. Исключения составляют класс Integer (примитивный тип int) и класс Character (примитивный тип char). Кроме содержания в себе значения, классы оболочки предоставляют обширный ряд методов, которые мы рассмотрим в этом уроке.

Объекты классов оболочек неизменяемые (immutable). Это значит, что объект не может быть изменен.

Все классы-обертки числовых типов имеют переопределенный метод equals(Object), сравнивающий примитивные значения объектов.

Оболочки типов

В следующей таблицы для каждого класса оболочки указан соответствующий примитивный тип и варианты конструкторов. Как вы видите каждый класс имеет два конструктора: один на вход принимает значение соответствующего примитивного значения, а второй - значение типа String. Исключения: класс Character, у которого только один конструктор с аргументом char и класс Float, объявляющий три конструктора - для значения float, String и еще double.

Оболочки типов

Примитивный тип	*Оболочка*	*Аргументы конструктора*
boolean	Boolean	boolean or String
byte	Byte	byte or String
char	Character	char
double	Double	double or String
float	Float	float, double, or String
int	Integer	int or String
long	Long	long or String
short	Short	short or String

Оболочки типов

Рассмотрим варианты вызова конструкторов на примере. Чтобы создать объект класса Integer, передаем в конструктор либо значение типа int либо String.

```
Integer i1 = new Integer(42);
Integer i2 = new Integer("42");

Float f1 = new Float(3.14f);
Float f2 = new Float("3.14f");

Character c1 = new Character('c');
```

Оболочки типов

Если передаваемая в конструктор строка не содержит числового значения, то выбросится исключение NumberFormatException.

При вызове конструктора с аргументом String класса Boolean, не обязательно передавать строки true или false. Если аргумент содержит любую другую строку, просто будет создан объект, содержащий значение false. Исключение выброшено не будет:

```
public class WrapperDemo1 {
    public static void main(String[] args) {
        Boolean boolean1 = new Boolean(true);
        Boolean boolean2 = new Boolean("Some String");

        System.out.println(boolean2);
    }
}
```

Оболочки типов

Как уже было сказано, классы оболочки содержат обширный ряд методов. Рассмотрим их.

Метод valueOf() предоставляет второй способ создания объектов оболочек. Метод перегруженный, для каждого класса существует два варианта - один принимает на вход значение соответствующего типа, а второй - значение типа String. Так же как и с конструкторами, передаваемая строка должна содержать числовое значение. Исключение составляет опять же класс Character - в нем объявлен только один метод, принимающий на вход значение char.

Оболочки типов

И в целочисленные классы Byte, Short, Integer, Long добавлен еще один метод, в который можно передать строку, содержащую число в любой системе исчисления. Вторым параметром вы указываете саму систему исчисления.

В следующем примере показано использование всех трех вариантов для создания объектов класса Integer:

```
public class WrapperValueOf {
    public static void main(String[] args) {
        Integer integer1 = Integer.valueOf("6");
        Integer integer2 = Integer.valueOf(6);
        // преобразовывает 101011 к 43
        Integer integer3 = Integer.valueOf("101011", 2);

        System.out.println(integer1);
        System.out.println(integer2);
        System.out.println(integer3);
    }
}
```

Оболочки типов

Методы parseXxx()

В каждом классе оболочке содержатся методы, позволяющие преобразовывать строку в соответствующее примитивное значение. В классе Double - это метод parseDouble(), в классе Long - parseLong() и так далее. Разница с методом valueOf() состоит в том, что метод valueOf() возвращает объект, а parseXxx() - примитивное значение.

Оболочки типов

Также в целочисленные классы Byte, Short, Integer, Long добавлен метод, в который можно передать строку, содержащую число в любой системе исчисления. Вторым параметром вы указываете саму систему исчисления. Следующий пример показывает использование метода parseLong():

```
public class WrapperDemo3 {
    public static void main(String[] args) {
        Long long1 = Long.valueOf("45");
        long long2 = Long.parseLong("67");
        long long3 = Long.parseLong("101010", 2);

        System.out.println("long1 = " + long1);
        System.out.println("long2 = " + long2);
        System.out.println("long3 = " + long3);
    }
}
```

Оболочки типов

Все типы-оболочки переопределяют toString(). Этот метод возвращает читабельную для человека форму значения, содержащегося в оболочке. Это позволяет выводить значение, передавая объект оболочки типа методу println():

```
Double double1 = Double.valueOf("4.6");
System.out.println(double1);
```

Также все числовые оболочки типов предоставляют статический метод toString(), на вход которого передается примитивное значение. Метод возвращает значение String:

```
String string1 = Double.toString(3.14);
```

Integer и Long предоставляют третий вариант toString() метода, позволяющий представить число в любой системе исчисления. Он статический, первый аргумент – примитивный тип, второй - основание системы счисления:

```
String string2 = Long.toString(254, 16); // string2 = "fe"
```

Оболочки типов

Integer и Long позволяют преобразовывать числа из десятичной системы исчисления к шестнадцатеричной, восьмеричной и двоичной. Например:

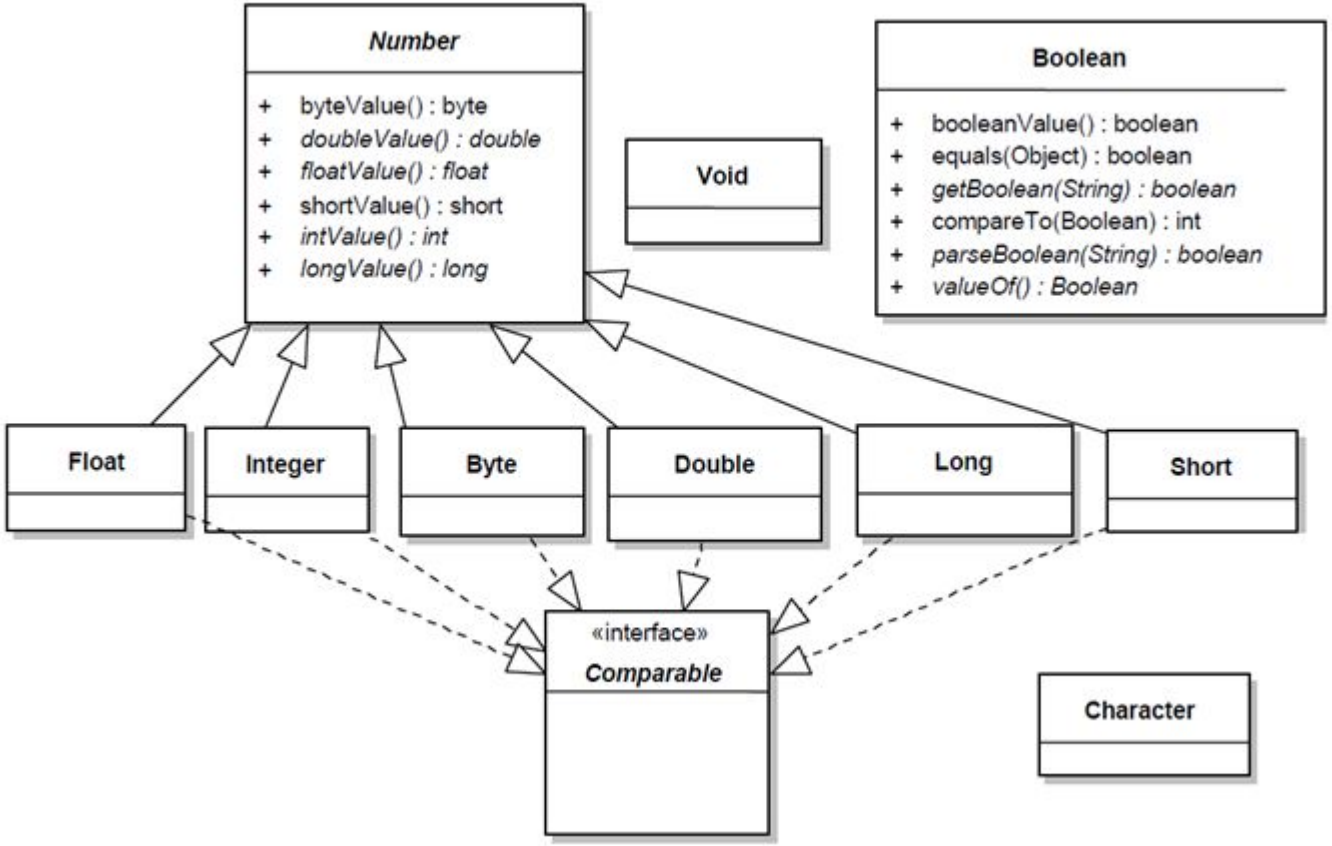
```
public class WrapperToXString {
    public static void main(String[] args) {
        String string1 = Integer.toHexString(254);
        System.out.println("254 в 16-ой системе = " + string1);

        String string2 = Long.toOctalString(254);
        System.out.println("254 в  8-ой системе = " + string2);

        String string3 = Long.toBinaryString(254);
        System.out.println("254 в  2-ой системе = " + string3);
    }
}
```

```
254 в 16-ой системе = fe
254 в  8-ой системе = 376
254 в  2-ой системе = 11111110
```

Оболочки типов



Все оболочки числовых типов наследуют абстрактный класс Number. Number объявляет методы, которые возвращают значение объекта в каждом из различных числовых форматов.

Оболочки типов

Пример приведения типов

```
public class WrapperDemo2 {
    public static void main(String[] args) {
        Integer iOb = new Integer(1000);
        System.out.println(iOb.byteValue());
        System.out.println(iOb.shortValue());
        System.out.println(iOb.intValue());
        System.out.println(iOb.longValue());
        System.out.println(iOb.floatValue());
        System.out.println(iOb.doubleValue());
    }
}
```

```
-24
1000
1000
1000
1000.0
1000.0
```

Оболочки типов

Каждый класс оболочка содержит статические константы, содержащие максимальное и минимальное значения для данного типа.

Например в классе Integer есть константы Integer.MIN_VALUE – минимальное int значение и Integer.MAX_VALUE – максимальное int значение.

Классы-обертки числовых типов Float и Double, помимо описанного для целочисленных примитивных типов, дополнительно содержат определения следующих констант:

- NEGATIVE_INFINITY – отрицательная бесконечность;
- POSITIVE_INFINITY – положительная бесконечность;
- NaN – не числовое значение (расшифровывается как Not a Number).

Оболочки типов

Следующий пример демонстрирует использование трех последних переменных. При делении на ноль возникает ошибка - на ноль делить нельзя. Чтобы этого не происходило, и ввели переменные NEGATIVE_INFINITY и POSITIVE_INFINITY. Результат умножения бесконечности на ноль - это значение NaN:

```
public class InfinityDemo {
    public static void main(String[] args) {
        int a = 7;
        double b = 0.0;
        double c = -0.0;
        double g = Double.NEGATIVE_INFINITY;
        System.out.println("7 / 0.0 = " + a / b);
        System.out.println("7 / -0.0 = " + a / c);
        System.out.println("0.0 == -0.0 = " + (b == c));
        System.out.println("-Infinity * 0 = " + g * 0);
    }
}
```

```
7 / 0.0 = Infinity
7 / -0.0 = -Infinity
0.0 == -0.0 =  true
-Infinity * 0 = NaN
```

Автоупаковка

Автоупаковка и распаковка это процесс преобразования примитивных типов в объектные и наоборот. Весь процесс выполняется автоматически средой выполнения Java (JRE).

```
public class AutoBoxDemo1 {
    public static void main(String[] args) {
        Integer iOb = 100; // упаковать значение int
        int i = iOb; // распаковать
        System.out.println(i + " " + iOb);
    }
}
```

Автоупаковка

Автоупаковка происходит при прямом присвоении примитива классу-обертке (с помощью оператора "="), либо при передаче примитива в параметры метода.

Распаковка происходит при прямом присвоении классу-обертке примитива.

Компилятор использует метод valueOf() для упаковки, а методы intValue(), doubleValue() и так далее, для распаковки.

Автоупаковка

Автоупаковка переменных примитивных типов требует точного соответствия типа исходного примитива — типу «класса-обертки».

Например, попытка автоупаковать переменную типа byte в Short, без предварительного явного приведения byte->short вызовет ошибку компиляции:

```
byte b = 4;
// Short s1 = b;
Short s2 = (short) b;
```

Автоупаковка

Автоупаковку можно использовать при вызове метода:

```
public class AutoBoxAndMethods {
    static int someMethod(Integer value) {
        return value;
    }

    public static void main(String[] args) {
        Integer iOb = someMethod(100);
        System.out.println(iOb);
    }
}
```

Автоупаковка

Внутри выражения числовой объект автоматически распаковывается. Выходной результат выражения при необходимости упаковывается заново:

```
public class AutoBoxAndOperations {
    public static void main(String[] args) {
        Integer iOb1, iOb2;
        int i;

        iOb1 = 100;

        iOb2 = iOb1 + iOb1 / 3;
        System.out.println("iOb2 после выражения: " + iOb2);

        i = iOb1 + iOb1 / 3;
        System.out.println("i после выражения: " + i);
    }
}
```

```
iOb2 после выражения: 133
i после выражения: 133
```

Автоупаковка

С появлением автоупаковки/распаковки стало возможным применять объекты Boolean для управления в операторе if и других циклических конструкциях Java:

```
public class AutoBoxAndCharacters {
    public static void main(String[] args) {
        Boolean b = true;

        if (b) {
            System.out.println("В if тоже можно использовать распаковку.");
        }

        Character ch = 'x';
        char ch2 = ch;

        System.out.println("ch2 = " + ch2);
    }
}
```

```
В if тоже можно использовать распаковку.
ch2 = x
```

Автоупаковка

До Java 5 работа с классами обертками была более трудоемкой:

```
public class AutoBoxDemo2 {
    public static void main(String[] args) {
        Integer y = new Integer(567);
        int x = y.intValue();
        x++;
        y = new Integer(x);
        System.out.println("y = " + y);
    }
}
```

```
y = 568
```

Автоупаковка

Перепишет тот же пример для работы с классами начиная с Java 5:

```
public class AutoBoxDemo3 {
    public static void main(String[] args) {
        Integer y = new Integer(567);
        y++;
        System.out.println("y = " + y);
    }
}
```

Автоупаковка

ВАЖНО! Объекты классов оболочек неизменяемые (immutable):

```
public class AutoBoxImmutability {
    public static void main(String[] args) {
        Integer y = 567;
        Integer x = y;
        // проверяем, что x и y указывают на один объект
        System.out.println(y == x);

        y++;
        System.out.println(x + " " + y);
        // проверяем, что x и y указывают на один объект
        System.out.println(y == x);
    }
}
```

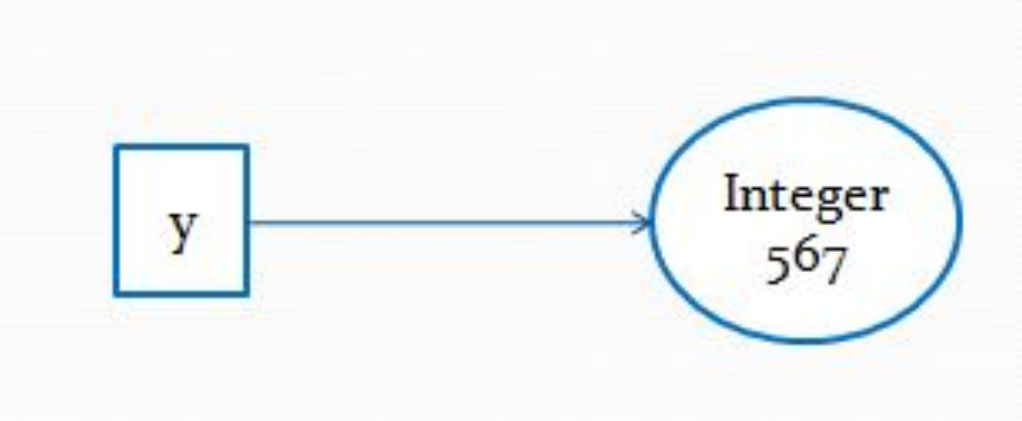
```
true
567 568
false
```

Автоупаковка

Рассмотрим следующий пример:

```
Integer y = 567;
```

Переменная y указывает на объект в памяти:

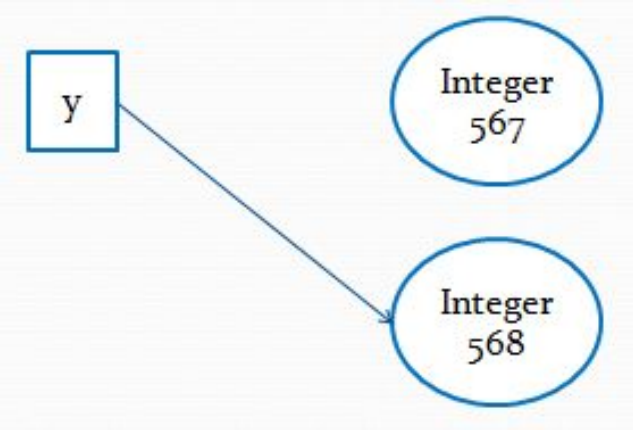


Автоупаковка

Если мы попытаемся изменить y, у нас создастся еще один объект в памяти, на который теперь и будет указывать y:

```
Integer y = 567;
y++;
```

Переменная y указывает на объект в памяти:



Автоупаковка

Кэширование объектов классов оболочек

Метод valueOf() не всегда создает новый объект. Он кэширует следующие значения:

- Boolean,
- Byte,
- Character от\u0000 до \u007f (7f это 127),
- Short и Integer от-128 до 127.

Если передаваемое значение выходит за эти пределы, то новый объект создается, а если нет, то нет.

Если мы пишем new Integer(), то гарантированно создается новый объект.

Автоупаковка

Рассмотрим это на следующем примере:

```
public class AutoBoxDemoCaching {
    public static void main(String[] args) {
        Integer i1 = 23;
        Integer i2 = 23;
        System.out.println(i1 == i2);
        System.out.println(i1.equals(i2));

        Integer i3 = 2300;
        Integer i4 = 2300;
        System.out.println(i3 == i4);
        System.out.println(i3.equals(i4));
    }
}
```

```
true
true
false
true
```

Перегрузка с дополнительными факторами

Перегрузка методов усложняется при одновременном использовании следующих факторов:

- расширение
- автоупаковка/распаковка
- аргументы переменной длины

Перегрузка с дополнительными факторами

```
public class EasyOver {
    static void go(int x) {
        System.out.print("int ");
    }

    static void go(long x) {
        System.out.print("long ");
    }

    static void go(double x) {
        System.out.print("double ");
    }

    public static void main(String[] args) {
        byte b = 5;
        short s = 5;
        long l = 5;
        float f = 5.0f;
        go(b);
        go(s);
        go(l);
        go(f);
    }
}
```

При расширение примитивных типов используется наименьший возможный вариант из всех методов.

```
int int long double
```

Перегрузка с дополнительными факторами

```
public class AddBoxing {
    public static void go(Integer x) {
        System.out.println("Integer");
    }

    public static void go(long x) {
        System.out.println("long");
    }

    public static void main(String[] args) {
        int i = 5;
        go(i); // какой go() вызовется?
    }
}
```

Расширение и boxing

Между расширением примитивных типов и boxing всегда выигрывает расширение. Исторически это более старый вид преобразования.

```
long
```

Перегрузка с дополнительными факторами

```
public class BoxAndWiden {
    public static void go(Object o) {
        Byte b2 = (Byte) o;
        System.out.println(b2);
    }

    public static void main(String[] args) {
        byte b = 5;
        go(b); // можно ли преобразовать byte в Object?
    }
}
```

Упаковка и расширение

Можно упаковать, а потом расширить. Значение типа int может стать Object, через преобразование Integer.

```
5
```

Перегрузка с дополнительными факторами

Расширение и упаковка

Нельзя расширить и упаковать. Значение типа byte не может стать Long. Нельзя расширить от одного класса обертки к другой. (IS-A не работает.)

```
public class WidenAndBox {
    static void go(Long x) {
        System.out.println("Long");
    }

    public static void main(String[] args) {
        byte b = 5;
        go(b); // нужно расширить до long и упаковать, что невозможно
    }
}
```

```
WidenAndBox.java:8: error: incompatible types: byte cannot be converted to Long
        go(b); // нужно расширить до long и упаковать, что невозможно
```

Перегрузка с дополнительными факторами

Между расширением примитивных типов и var-args всегда проигрывает var-args:

```
public class AddVarargs {
    public static void go(int x, int y) {
        System.out.println("int,int");
    }

    public static void go(byte... x) {
        System.out.println("byte... ");
    }

    public static void main(String[] args) {
        byte b = 5;
        go(b, b); // какой go() вызовется?
    }
}
```

```
int,int
```

Перегрузка с дополнительными факторами

Упаковка и var-args совместимы с перегрузкой методов. Var-args всегда проигрывает:

```
public class BoxOrVararg {
    public static void go(Byte x, Byte y) {
        System.out.println("Byte, Byte");
    }

    public static void go(byte... x) {
        System.out.println("byte... ");
    }

    public static void main(String[] args) {
        byte b = 5;
        go(b, b); // какой go() вызовется?
    }
}
```

```
Byte, Byte
```

Перегрузка с дополнительными факторами

Подытожим все правила:

1. При расширение примитивных типов используется наименьший возможный вариант из всех методов.
2. Между расширением примитивных типов и упаковкой всегда выигрывает расширение. Исторически это более старый вид преобразования.
3. Можно упаковать, а потом расширить. (Значение типа int может стать Object, через преобразование Integer.)
4. Нельзя расширить и упаковать. Значение типа byte не может стать Long. Нельзя расширить от одного класса обертки к другой. (IS-A не работает.)
5. Можно комбинировать var-args с расширением или упаковкой. var-args всегда проигрывает.