常用类-day01

今日学习内容

- 工具类的设计
- 单例模式
- 八大基本类型的包装类
- 装箱和拆箱
- 包装类的缓存设计
- BigDecimal类
- String类
- StringBuilder和StringBuffer类

今日学习目标

- 熟悉查看API，熟悉方法调用
- 了解工具的两种设计方案，公共静态方法和单例模式
- 掌握单例模式的编写
- 掌握八大基本数据类型的包装类
- 了解基本类型和包装类的区别
- 掌握什么是装箱和拆箱，什么是自动装箱和拆箱
- 掌握BigDecimal的加减乘除和保留精度操作
- 掌握String常用方法
- 掌握StringBuilder的操作
- 掌握String、StringBuilder、StringBuffer三者的区别

1.1 工具类的设计

一般地，把那些完成**通用功能的方法**分类存放到类中，这些类就叫工具类。

- 工具类起名：XxxUtil、XxxUtils、XxxTool、XxxTools等，其中Xxx表示一类事物，比如ArrayUtil、StringUtil、JdbcUtil。
- 工具类存放的包起名：util、utils、tool、tools等
- 工具类在开发中的应用场景 ： 作为工具性质且能高效地重复使用。

工具类如何设计，在开发中有两种设计：

- 如果工具方法全部使用public static修饰
  - 此时只需要使用工具类名调用工具方法
  - 此时必须把工具类的构造器私有化，防止创建工具类的对象来调用静态方法
- 如果工具方法没有使用static修饰
  - 此时必须使用工具类的对象去调用工具方法
  - 此时把必须工具类设计为**单例模式**的

一般的出于简单考虑，首选第一种，如JDK中提供的工具java.util.Arrays类。

1.1.1 公共静态方法（掌握）

比如使用公共静态方法的方式，设计一个数组的工具类。

```
public class ArrayUtil {

    // 1> 构造器私有化
    private ArrayUtil () { }

    // 2> 提供public static 方法作为工具方法
    public static String array2String(int[] array) {
        String str = "[";
        int item;
        for(int i = 0;i < array.length;i++) {
            item = array[i];
            if( i == array.length - 1 ) {
                str += (item + "]");
            }else {
                str += (item + ",");
            }
        }
        return str;
    }
}
```

调用者直接使用 `工具类名.工具方法名称` 完成调用：

```
int[] arr = new int[]{7,4,2,8,1,9};
System.out.println(ArrayUtil.array2String(arr));
```

1.1.2 单例模式（掌握）

设计模式（Design pattern）：是一套被反复使用的代码设计经验总结，专门用于解决特定场景的需求。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。

比如使用单例模式的方式，设计一个数组的工具类。

单例设计模式（singleton）：最常用、最简单的设计模式，单例的编写有N种写法。

目的：保证在整个应用中某一个类有且只有一个实例（一个类在堆内存只存在一个对象）。

1.1.2.1 饿汉式

[1] 必须在该类中，自己先创建出一个对象

[2] 私有化自身的构造器，防止外界通过构造器创建新的工具类对象

[3] 向外暴露一个公共的静态方法用于返回自身的对象

```
package cn.wolfcode02.singleton;

// 单例模式（饿汉式）
public class ArrayUtils {
```

```
    // [1] 事先创建好当前类的一个对象
    private  static ArrayUtils instance = new ArrayUtils();

    // [2]私有化构造方法。
    private ArrayUtils(){}

    // [3] 提供一个公共静态方法(用于统一的外界访问方式)，返回事先创建好的实例
    public static  ArrayUtils getInstance(){
        return instance;
    }

    // 把工具方法作为实例方法附着到单例上
    public String array2String(int[] array){
        String str = "[";
        int item = array[i];
        for(int i = 0;i < array.length;i++){
            if(i == array.length - 1){
                str += item + "]";
            }else{
                str += item + ",";
            }
        }
        return str;
    }
}
```

创建测试类测试单例设计模式

```
package cn.wolfcode02.singleton;

public class Test01 {
    public static void main(String[] args) {
        // 1> 单例访问
        ArrayUtils instance2 = ArrayUtils.getInstance();
        ArrayUtils instance3 = ArrayUtils.getInstance();
        System.out.println("instance2 = " + instance2);
        System.out.println("instance3 = " + instance3);

        // 2> 测试工具方法
        ArrayUtils instance = ArrayUtils.getInstance();
        int[] arr = {1,2,3,4};
        String str = instance.array2String(arr);
        System.out.println("str = " + str);

    }
}
```

1.1.2.2 懒汉式

```
// 单例模式（饿汉式）
public class ArrayUtils {
    // [1]私有化构造方法。
    private ArrayUtils(){}

    // [2] 事先创建好当前类的一个对象
    private  static ArrayUtils instance = null;
```

```
    // [3] 提供一个公共静态方法(用于统一的外界访问方式)，返回事先创建好的实例
    public static  ArrayUtils getInstance(){
        if(null == instance){
            instance = new ArrayUtils();
        }
        return instance;
    }

    // 工具方法
    public String array2String(int[] array){
        String str = "[";
        int item = array[i];
        for(int i = 0;i < array.length;i++){
            if(i == array.length - 1){
                str += item + "]";
            }else{
                str += item + ",";
            }
        }
        return str;
    }
}
```

总结

懒汉式和饿汉式单例设计模式的区别?

1.1.2.4 枚举法

```
public enum ArrayUtilsEnum {
    // [1]. 定义枚举常量
    INSTANCE;

    // [2] 定义工具静态方法
    public String array2String(int[] array){
        String str = "[";
        int item = array[i];
        for(int i = 0;i < array.length;i++){
            if(i == array.length - 1){
                str += item + "]";
            }else{
                str += item + ",";
            }
        }
        return str;
    }
}
```

创建测试类测试单例设计模式

```
public class Test01 {
    public static void main(String[] args) {
        int[] arr = {5,3,6,2};
        // 如何测试 ?
    }
}
```

> 常用类怎么去学习?

```
常用类是java jdk 提供的直接可以使用的类，一般而言，
常用类已经包含了面向对象的所有基础知识，作为初学者可以看到面向对象思想的应用。

既然常用类已经给开发者写好了，作为开发者或者初学者怎么去学习呢？
一个类可以创建对象，该对象一定具备类中定义的功能，面向对象主张 "让谁（具体对象）去做"
实际开发过程中，一定要想着找合适的对象解决你的问题，所以
一定要用一句话总结一个类的功能，同时知晓该类中定义的常用方法。这是初学者最基本的要求。
如果有多余的时间，最好把该类涉及的底层实现原理通过团队(小组)来解决。
```

1.2 包装类

1.2.1 为什么要存在包装类（了解）

思考 1: 使用int基本类型来表示学生的考试成绩，此时怎么区分考试成绩为0和没有成绩两种情况?使用int是不行的，只能表示0的情况，此时要解决该问题就得使用基本类型的包装类。

思考2：经常性地，我们需要把字符串转化成int类型，难道每次都自己写算法把字符串转化成int类型吗?

在实际开发过程中，我们经常性的会利用`封装`的思维把一些数据封装到类中，并提供方法对这些数据进行操作。

需求：模拟定义一个类来封装int类型的值。

```
public  class IntWrapper {
    private int value;  // 封装的字段值

    public  IntWrapper(int value) {
         this.value = value;
    }
}
```

使用该int的包装类。

```
IntWrapper wrapper = null;              //没有对象,没有数据
IntWrapper wrapper = new IntWrapper(0); //有对象，表示数据0
```

```
public class IntWrapper {

    private int value;

    public IntWrapper(int value) {
        this.value = value;
    }
```

```
    public String toString() {
        return this.value + "";
    }

    public int intValue() {
        return this.value;
    }

    public float floatValue() {
        return this.value * 1.0f;
    }

    public double doubleValue() {
        return this.value * 1.0;
    }
}
```

总结:

1. 模拟的int包装类IntWrapper既可以表示0，也可以表示null。
2. IntWrapper 提供了方法用于对封装的value进行进一步的操作

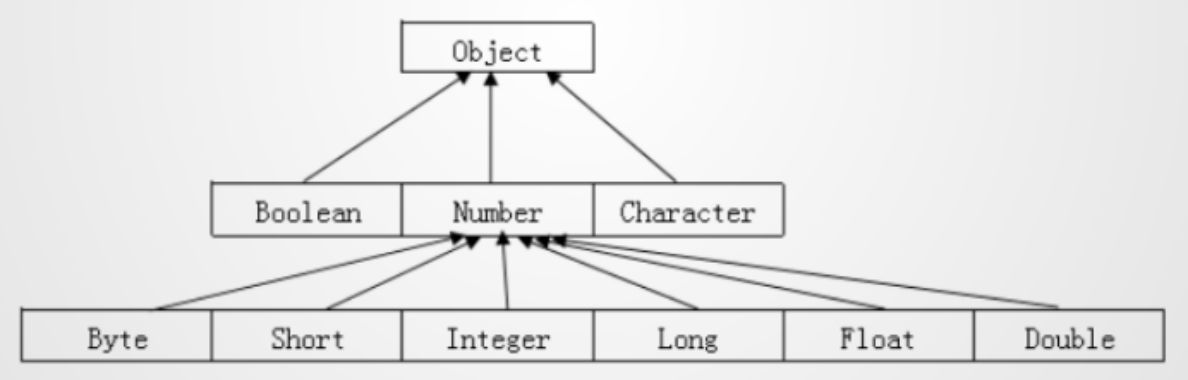
1.2.2 包装类

1.2.2.1 包装类概述（了解）

包装类就是把基本数据类型封装到一个类中，提供便利的方法，让开发者更方便的操作基本类型。

包装类的出现不是为了取代基本数据类型。

包装类位于 `java.lang` 包中



1.3 Integer(掌握)

Integer内部封装了一个int类型的基本数据类型value，并提供了方法对int值进行操作，还提供了int值和String之间的转换。

1.3.2 常用方法(掌握)

```
public static void main(String[] args) {
    // 传入一个int值构建一个Integer对象，此构造方法不推荐使用
    Integer i1 = new Integer(10);
```

```
    // 推荐使用
    Integer i2 = Integer.valueOf(10);

    // Integer 可以把String -> int
    Integer i3 = Integer.valueOf("100");
    int num = i3.intValue();
    System.out.println(num);

    // 实际开发过程中
    int num2 = Integer.parseInt("200");
    System.out.println(num2);
}
```

以上通过通过Integer把字符串直接转化成基本数据类型int了，也可以通过Integer把int转化成String

```
public static void main(String[] args) {

    // int -> String
    Integer i1 = Integer.valueOf(10);
    String str2 = i1.toString();
    System.out.println(str2);

    // 实际开发过过程中 int -> String
    String str3 = Integer.toString(10);

    // 更推荐语法糖
    int num = 10;
    String str = num + "";

    /**
    * 总结
    * Integer 作为一个中间桥梁 int <- Integer -> String
    */
}
```

1.4 Auto-Boxing 和 Auto-UnBoxing

1.4.1 装箱和拆箱（掌握）

装箱：把基本类型数据转成对应的包装类对象。
拆箱：把包装类对象转成对应的基本数据类型。

装箱操作：

```
方式一：    Integer num1 = new Integer(17);
方式二：    Integer num2 = Integer.valueOf(17); //建议
```

拆箱操作：

```
Integer num3  =  Integer.valueOf(17);   //装箱操作
int val = num3.intValue();  //拆箱操作
```

从Java5开始提供了的自动装箱（AutoBoxing）和自动拆箱（AutoUnBoxing）功能：

自动装箱：可把一个基本类型变量直接赋给对应的包装类变量。

自动拆箱：可以把包装类对象直接赋给对应的基本数据类型变量。

```
Integer num4  =  17;     // 装箱操作
int val2 = num4;         // 拆箱操作
```

自动装箱和拆箱，在底层依然是手动装箱和拆箱。

思考Object obj = 17;代码正确吗？为什么？

```
Integer i = 17;      //自动装箱操作
Object obj = i;      //把子类对象赋给父类变量
```

1.4.2 缓存设计（了解）

从性能上考虑，把常用数据存储到缓存区域，使用时不需要每次都创建新的对象，可以提高性能。

- Byte、Short、Integer、Long：缓存范围[-128，127]；
- Character：缓存范围[0，127]；

```
Integer i1 = new Integer(123);
Integer i2 = new Integer(123);
System.out.println(i1 == i2);// false

Integer i3 = Integer.valueOf(123);
Integer i4 = Integer.valueOf(123);
System.out.println(i3 == i4);// true

Integer i5 = 123; // 底层等价于
Integer i6 = 123;
System.out.println(i5 == i6);// true
```

Integer的部分源代码如下：

```java
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

```java
private static class IntegerCache {
    static final int low = -128;
    static final int high;
    static final Integer cache[];

    static {
        // high value may be configured by property
        int h = 127;
        String integerCacheHighPropValue =
            sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
        if (integerCacheHighPropValue != null) {
            int i = parseInt(integerCacheHighPropValue);
            i = Math.max(i, 127);
            // Maximum array size is Integer.MAX_VALUE
            h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
        }
        high = h;

        cache = new Integer[(high - low) + 1];
        int j = low;
        for(int k = 0; k < cache.length; k++)
            cache[k] = new Integer(j++);

        // range [-128, 127] must be interned (JLS7 5.1.7)
        assert IntegerCache.high >= 127;
    }

    private IntegerCache() {}
}
```

如果把上述代码中的123换成250，则结果都为false。

1.3 BigDecimal（掌握）

float和double都不能表示精确的小数，使用BigDecimal类可以解决该问题，BigDecimal用于处理金钱或任意精度要求高的数据。

1.3.1 基本运算

BigDecimal不能直接把赋值和运算操作，只能通过构造器传递数据，而且必须使用**字符串类型的构造器**，操作BigDecimal主要是加减乘除四个操作。

```
// 使用double类型：
System.out.println(0.09 + 0.01);// ?

// 使用BigDecimal类型double类型的构造器：
BigDecimal num1 = new BigDecimal(0.09);
BigDecimal num2 = new BigDecimal(0.01);
System.out.println(num1.add(num2));// ?

// 使用BigDecimal类型String类型的构造器：
BigDecimal num3 = new BigDecimal("0.09");
BigDecimal num4 = new BigDecimal("0.01");
System.out.println(num3.add(num4));// ?
```

结果为：

```
0.09999999999999999
0.099999999999999999687749774324174723005853593349456787109375
0.10
```

1.3.2 精度控制 和 除不尽问题

```
public static void main(String[] args) {
    //
    BigDecimal num1 = new BigDecimal("10.0");
    BigDecimal num2 = new BigDecimal("3.0");

    // 1. 保留位数和精度控制
    // RoundingMode 舍入模式
    // RoundingMode.HALF_UP      四舍五入
    // RoundingMode.HALF_Down    四舍六入
    BigDecimal r1 = num1.multiply(num2).setScale(2,RoundingMode.HALF_UP);
    System.out.println("r1 = " + r1);

    // 2. 除不尽问题
    // java.lang.ArithmeticException
    // Non-terminating decimal expansion; no exact representable decimal result.
    // 报错原因:除不尽(3.333333333...333...)
    BigDecimal r2 = num1.divide(num2,3,RoundingMode.HALF_UP);
    System.out.println("r2 = " + r2);
}
```

上述代码分别表示乘法和除法按照四舍五入方式保留两位小数。

1.4 String（掌握）

字符串（字符序列），表示把多个字符按照一定得顺序连成的字符序列。

字符串的分类（根据同一个对象，内容能不能改变而区分）：

- **不可变的字符串——String**：当String对象创建完毕之后，该对象的内容是不能改变的，一旦内容改变就变成了一个新的对象。

- **可变的字符串——StringBuilder/StringBuffer**：当StringBuilder对象创建完毕之后，对象的内容可以发生改变，当内容发生改变的时候，对象保持不变。

1.4.1 String 本质概述(了解)

String 类型表示字符串，Java 程序中的所有字符串字面值（如 "abc" ）都作为此类的实例实现。

String 在内存中是以**字符数组**的形式存在

```
// jdk1.8 and before
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final char value[];

    ...
}
```

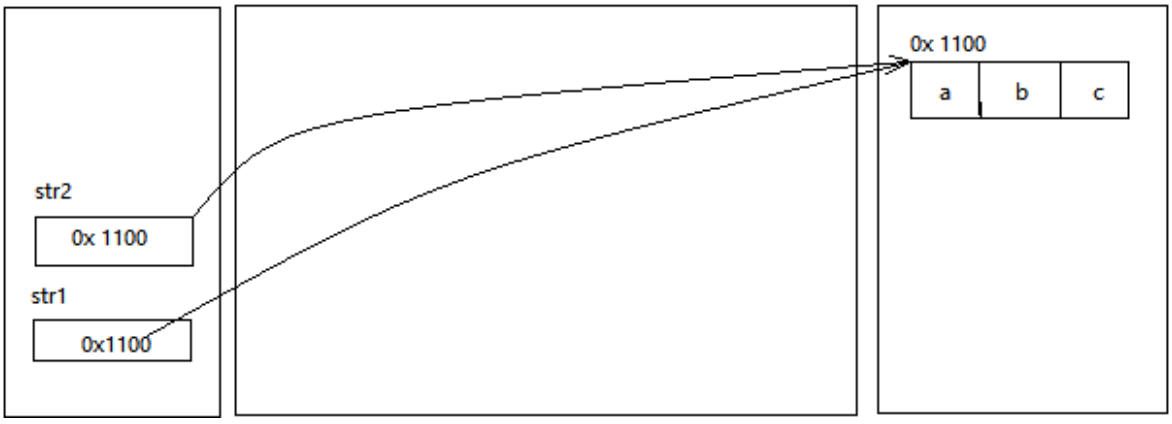
我们可以认为，**String对字符数组的封装**，并提供以只读的形式操作其封装的字符数组的方法。

String对象的创建的两种方式：

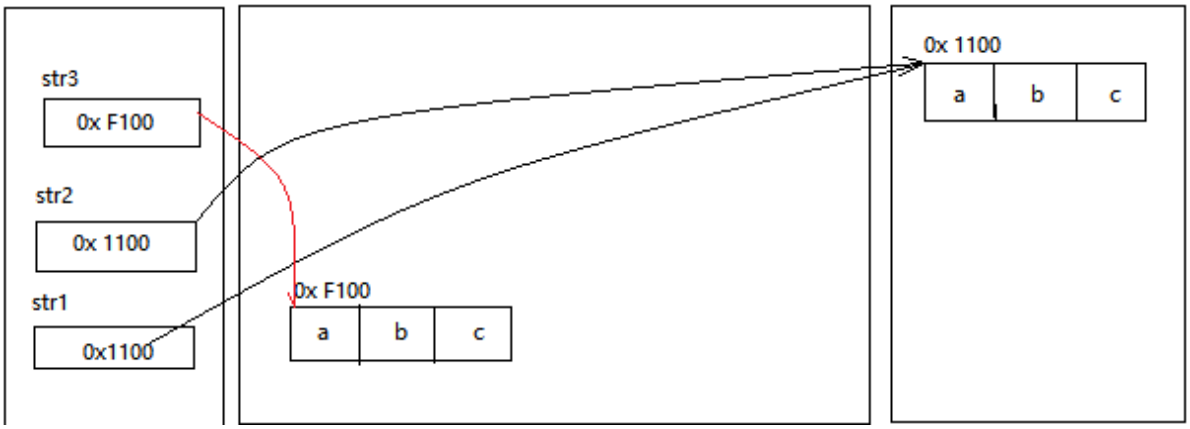
```
1、直接赋一个字面量:        String   str1  =  "ABCD";//直接存储在方法区的常量池中,节约内存
2、通过构造器创建:          String   str2  =  new String("ABCD");
```

字符串内存图

通过**字面量创建的字符串分配在常量池**中，所以字面量字符串是常量；它们的值在创建之后不能更改，因为 String 对象是不可变的，所以可以共享。

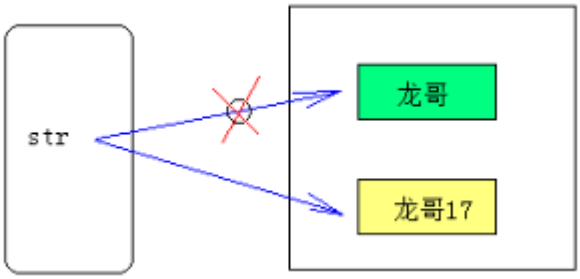


通过new 操作创建的字符串分配在堆区。



String类，表示不可变的字符串，当String对象创建完毕之后，该对象的内容是不能改变的，一旦内容改变就变成了一个新的对象，看下面代码。

```
String str = "龙哥";
str = str + "hello";
```



String对象的 "空" 值

```
表示引用为空(null)
String  str1 = null;   //没有初始化，没有分配内存空间.

内容为空字符串
String  str2  = "";     // 已经初始化，分配内存空间，不过没有内容
```

1.4.3 字符串常用方法(掌握)

"ABCD" ['A','B','C,'D']

- int length() 返回此字符串的字符个数
- char charAt(int index) 返回指定索引位置的字符
- int indexOf(String str) 返回指定字符串在此字符串中从左向右第一次出现处的索引位置
- boolean equals(Object anObject) 比较内容是否相同
- boolean equalsIgnoreCase(String anotherString) 忽略虑大小写，比较内容是否相同
- String toUpperCase() 把当前字符串转换为大写
- String toLowerCase() 把当前字符串转换为小写
- String substring(int beginIndex)：从指定位置开始截取字符串
- String substring(int beginIndex, int endIndex)：截取指定区域的字符串
- boolean endWith(String suffix)
- boolean startWith(String prefix)
- replace(char oldChar，char newChar)

需求：判断字符串非空：字符串不为null并且字符内容不能为空字符串("")

判断一个字符串非空：

```
public static boolean hasLength(String str) {
    return str != null && ! "".equals(str.trim());
}
```

总结:

【1】通过这些方法，你发现了什么?

1.4.4 字符串陷阱(了解)

思考一下问题：

"A" + "B" 在内存中创建了几个字符串?

"A" + "B" + "C" 呢?

实际开发过程中，一定要规避以下代码

```
String tmp = "";
for(int i=0;i<5;i++){
    tmp += "hello";
}
```

以上代码存在什么问题？（提问学生）