

枚举

- 1) 枚举对应英文 (enumeration , 简写 enum)
- 2) 枚举是一组常量的集合
- 3) 可以理解为: 枚举属于一种特殊的类, 里面只包含一组有限的特定的对象

自定义枚举

1. 将构造器私有化, 目的是防止 直接 new
2. 去掉 setXxx 相关的方法, 防止属性被修改
3. 在类的内部, 直接创建固定的对象

```
//演示定义枚举实现
class Season { //类
    private String name;
    private String desc; //描述

    //定义了四个对象
    public static Season SPRING = new Season("春天", "温暖");
    public static Season WINTER = new Season("冬天", "寒冷");
    public static Season AUTUMN = new Season("秋天", "凉爽");
    public static Season SUMMER = new Season("夏天", "炎热");
}
```

4. 优化, 可以再加入 final 修饰符
5. 对枚举对象/属性使用 final + static 共同修饰, 实现底层优化
6. 枚举对象名通常使用全部大写, 常量的命名规范
7. 枚举对象根据需要也可以有多个属性

特点:

- 1) 构造器私有化
- 2) 本类内部创建一组对象
- 3) 对外暴露对象 (通过为对象添加 public final static 修饰符)
- 4) 可以提供 get 方法, 但不要提供 set 方法

enum 关键字实现枚举的注意事项:

- 1) 当我们使用 enum 关键字开发一个枚举类时, 默认会继承 enum 类
- 2) 传统的 public static final Season2 SPRING = new Season2("春天", "温暖"); 简化成 SPRING("春天", "温暖"), 这里必须知道它调用的是哪个构造器
- 3) 如果使用无参构造器创建枚举对象, 则实参列表和小括号都可以省略
- 4) 当有多个枚举对象时, 使用 , 间隔, 最后有一个分号结尾
- 5) 枚举对象必须放在枚举类的行首

enum 常用方法说明: 使用关键字 enum 时, 会隐式继承 enum 类, 这样就可以使用 enum 类相关的方法。

```
public abstract class Enum<E extends Enum<E>>
    implements Comparable<E>, Serializable{
}
}
```

enum常用方法应用实例

我们一起来举例说明enum常用的方法的使用，对Season2测试。EnumMethod.java

1. toString: Enum类已经重写过了，返回的是当前对象名，子类可以重写该方法，用于返回对象的属性信息
2. name: 返回当前对象名（常量名），子类中不能重写
3. ordinal: 返回当前对象的位置号，默认从0开始
4. values: 返回当前枚举类中所有的常量
5. valueOf: 将字符串转换成枚举对象，要求字符串必须为已有的常量名，否则报异常！
6. compareTo: 比较两个枚举常量，比较的就是位置号！

```
Season2 autumn = Season2.AUTUMN;
System.out.println(autumn);
System.out.println(autumn.name());
System.out.println(autumn.ordinal()); //2
Season2[] values = Season2.values();
for (Season2 season2 : values) { //增强for
    System.out.println(season2);
}
Season2 value = Season2.valueOf("SPRING");
System.out.println(value);
System.out.println(autumn.compareTo(value));
```

enum 实现接口：

- 1) 使用 enum 关键字后就不能再继承其他类了，因为 enum 会隐式继承 Enum，而 Java 是单继承机制
- 2) 枚举类和普通类一样，可以实现接口，如下形式：
enum 类名 implements 接口1, 接口2{}

注解的理解

- 1) 注解（Annotation）也被称为元数据（Metadata），用于修饰解释 包、类、方法、属性、构造器、局部变量等数据信息
- 2) 和注释一样，注解不影响程序逻辑，但注解可以被编译或运行，相当于嵌入在代码中的补充信息
- 3) 在 JavaSE 中，注解的使用目的比较简单，例如标记过时的功能，忽略警告等。在 JavaEE 中注解占据了更重要的角色，例如用来配置应用程序的任何切面，代替 Java EE 旧版中所遗留的繁冗代码和 XML 配置等

基本的 Annotation 介绍

- 使用 Annotation 时要在其前面增加 @ 符号，并把该 Annotation 当成一个修饰符使用。用于修饰它支持的程序元素
- 三个基本的 Annotation:
 - 1) @Override: 限定某个方法，是重写父类方法，该注解只能用于方法
 - 2) @Deprecated: 用于表示某个程序元素(类, 方法等)已过时
 - 3) @SuppressWarnings: 抑制编译器警告

基本的 Annotation应用案例

➤ Override 使用说明

1. @Override 表示指定重写父类的方法（从编译层面验证），如果父类没有fly方法，则会报错
2. 如果不写@Override 注解，而父类仍有 public void fly(){}，仍然构成重写
3. @Override 只能修饰方法，不能修饰其它类，包，属性等等
4. 查看@Override注解源码为 @Target(ElementType.METHOD),说明只能修饰方法
5. @Target 是修饰注解的注解，称为元注解

基本的 Annotation应用案例

• @SuppressWarnings 注解的案例 SuppressWarnings_.java

@SuppressWarnings: 抑制编译器警告

➤ 案例

```
public class AnnotationDemo01 {  
    public static void main(String[] args) {  
        List list = new ArrayList();  
        list.add("");  
        list.add("");  
        list.add("");  
        int i;  
        System.out.println(list.get(1));  
    }  
}
```

基本的 Annotation应用案例

• @SuppressWarnings 注解的案例

➤ 说明各种值

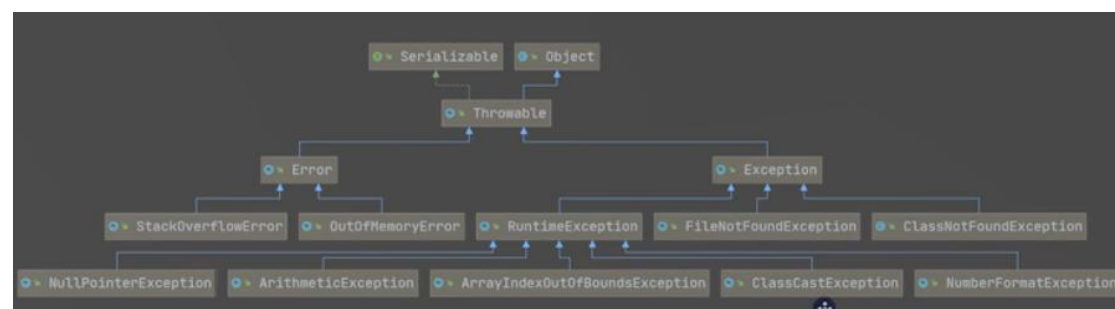
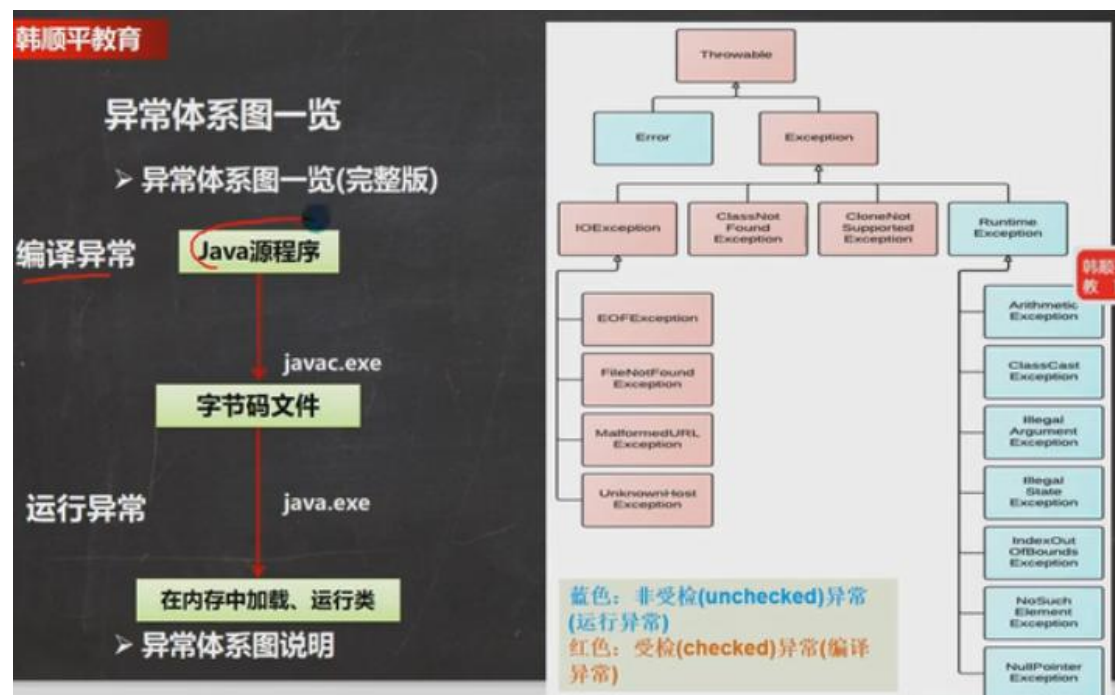
- 1) unchecked 是忽略没有检查的警告
- 2) rawtypes 是忽略没有指定泛型的警告(传参时没有指定泛型的警告错误)
- 3) unused 是忽略没有使用某个变量的警告错误
- 4) @SuppressWarnings 可以修饰的程序元素为，查看@Target
- 5) 生成@SupperssWarnings 时，不用背，直接点击左侧的黄色提示，就可以选择(注意可以指定生成的位置)

Java 语言中，将程序执行中发生的不正常情况称为“异常”（开发过程中的语法错误和逻辑错误不是异常）

执行过程中所发生的异常事件可以分为两类：

- 1) Error(错误): java 虚拟机无法解决的严重问题，如：JVM 系统内部错误、资源耗尽等严重情况。如：StackOverflowError[栈溢出]和 OOM(out of memory)，Error 是严重错误，程序会崩溃；
- 2) Exception(异常): 其他因编程错误或偶然的外在因素导致的一般性问题，可以使用针对性的代码进行处理。例如空指针访问，试图读取不存在的文件，网络连接中断等，Exception 分为两大类：运行时异常[程序运行时发生的异常]和编译时异常[编译时，编译器检查出的异常]。

异常的体系图：



类的异常体系图，体现了继承和实现的关系

● 异常体系图的小结

1. 异常分为两大类，运行时异常和编译时异常。
2. 运行时异常，编译器不要求强制处置的异常。一般是指编程时的逻辑错误，是程序员应该避免其出现的异常。java.lang.RuntimeException类及它的子类都是运行时异常
3. 对于运行时异常，可以不作处理，因为这类异常很普遍，若全处理可能会对程序的可读性和运行效率产生影响
4. 编译时异常，是编译器要求必须处置的异常。

常见的运行时异常

● 常见的运行时异常包括

- 1) NullPointerException空指针异常
- 2) ArithmeticException数学运算异常
- 3) ArrayIndexOutOfBoundsException数组下标越界异常
- 4) ClassCastException类型转换异常
- 5) NumberFormatException数字格式不正确异常[]

- 1) NullPointerException 空指针异常：当应用程序试图在需要对象的地方使用 null 时，抛出该异常；
- 2) ArithmeticException 数学运算异常：当出现异常的运算条件时，抛出此异常。例如，一个整数“除以零”时，抛出此类的一个实例；
- 3) ArrayIndexOutOfBoundsException 数组下标越界异常：用非法索引访问数组时抛出的异常，如果索引为负或大于等于数组大小，则该索引为非法索引；
- 4) ClassCastException 类型转换异常：当试图将对象强制转换成为不是实例的子类时，抛出该异常，例如：以下代码将生成一个 ClassCastException；
- 5) NumberFormatException 数字格式不正确异常：当应用程序试图将字符串转换成一个数值类型，但该字符串不能转换为适当格式时，抛出该异常=>使用异常我们可以确保输入是满足条件的数字。

编译异常：是指在编译期间，就必须处理的异常，否则代码不能通过编译
常见的编译异常：

● 常见的编译异常

- ✓ SQLException //操作数据库时，查询表可能发生异常
- ✓ IOException //操作文件时，发生的异常
- ✓ FileNotFoundException //当操作一个不存在的文件时，发生异常
- ✓ ClassNotFoundException //加载类，而该类不存在时，异常
- ✓ EOFException // 操作文件，到文件末尾，发生异常
- ✓ IllegalArgumentException //参数异常

异常处理：异常处理就是当异常发生时对异常的处理方式

- 1) try-catch-finally：程序员在代码中捕获发生的异常，自行处理
- 2) throws：将发生的异常抛出，交给调用者（方法）来处理，最顶级的处理者是 JVM（如果程序员没有显式得处理异常，那默认用 throws）

try-catch异常处理

- try-catch方式处理异常-注意事项 **TryCatchDetail.**

- 1) 如果异常发生了, 则异常发生后面的代码不会执行, 直接进入catch块.
- 2) 如果异常没有发生, 则顺序执行try的代码块, 不会进入到catch.
- 3) 如果希望不管是否发生异常, 都执行某段代码(比如关闭连接, 释放资源等) 则使用如下代码- finally { }

```
try{  
    //可疑代码  
}catch(异常){  
    //....  
}finally{  
    //释放资源等..  
}
```

```
try {  
    int a = Integer.parseInt(str);  
    System.out.println("数字: " + a);  
} catch (Exception e) {  
    e.printStackTrace();  
} finally {  
    System.out.println(" 不管是否发生异常, 始终执行的代码~~");  
}
```

try-catch异常处理

- 4) 可以有多个catch语句, 捕获不同的异常(进行不同的业务处理), 要求父类异常在后, 子类异常在前, 比如(Exception 在后, NullPointerException 在前), 如果发生异常, 只会匹配一个catch, 案例演示

```
try {  
}catch(NullPointerException e) {  
}catch(Exception e) {  
}finally{  
}
```

```
try {  
    Person person = new Person();  
    person.say();  
    int i = 10 / 0;  
} catch (NullPointerException e) {  
    System.out.println("here1");  
    e.printStackTrace();  
} catch (Exception e) {  
    System.out.println("here2");  
    e.printStackTrace();  
} finally {  
    System.out.println("here3");  
}
```

try-catch异常处理

- 5) 可以进行 try-finally 配合使用, 这种用法相当于没有捕获异常, 因此程序会直接崩掉。

```
try {  
    //代码...  
}  
finally{ //总是执行  
}
```

● try-catch-finally执行顺序小结

- 1) 如果没有出现异常, 则执行try块中所有语句, 不执行catch块中语句, 如果有finally, 最后还需要执行finally里面的语句
- 2) 如果出现异常, 则try块中异常发生后, 剩下的语句不再执行。将执行catch块中的语句, 如果有finally, 最后还需要执行finally里面的语句!

throws 异常处理:

- 1) 如果一个方法中的语句执行时可能生成某种异常, 但是不能确定如何处理这种异常, 则此方法应显式地声明抛出异常, 表明该方法将不对这些异常进行处理, 而由该方法的调用者负责处理;
- 2) 在方法声明中用 throws 语句可以声明抛出异常的列表, throws 后面的异常类型可以是方法中产生的异常类型, 也可以是它的父类。

throws 异常处理:

- 1) 对于编译异常, 程序中必须处理, 比如 try-catch 或者 throws
- 2) 对于运行时异常, 程序中如果没有处理, 默认就是 throws 的方法处理
- 3) 子类重写父类的方法时, 对抛出异常的规定: 子类重写的方法, 所抛出的异常类型要么和父类抛出的异常一致, 要么为父类抛出的异常的类型子类型
- 4) 在 throws 过程中, 如果有方法 try-catch, 就相当于异常处理, 就可以不必 throws
- 5) 编译类型的异常必须抛出或者解决, 但是运行类型的异常有默认解决机制

自定义异常

● 自定义异常的步骤

- 1) 定义类: 自定义异常类名(程序员自己写) 继承Exception或RuntimeException
- 2) 如果继承Exception, 属于编译异常
- 3) 如果继承RuntimeException, 属于运行异常(一般来说, 继承RuntimeException)

	意义	位置	后面跟的东西
throws	异常处理的一种方式	方法声明处	异常类型
throw	手动生成异常对象的关键字	方法体中	异常对象

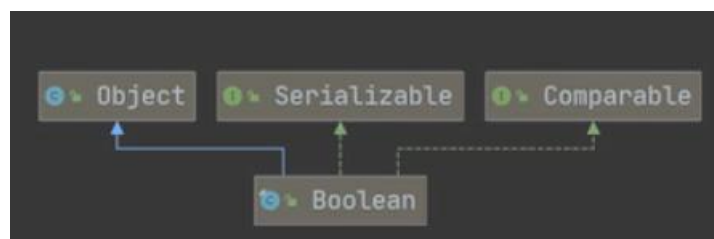
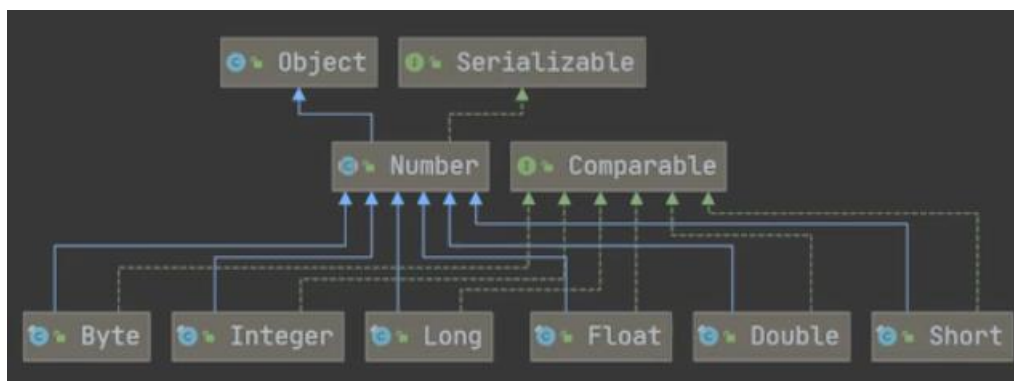
常用类:

包装类的分类:

1. 针对八种基本数据类型相应的引用类型——包装类;
2. 有了类的特点, 就可以调用类的方法

基本数据类型	包装类
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

父类: Number



ctrl + atl + t 在选中一段代码后可以直接实现 try-catch 语句

包装类和基本数据的转化:

演示 包装类 和 基本数据类型的相互转换, 这里以int 和 Integer演示。
 1) jdk5 前的手动装箱和拆箱方式, 装箱: 基本类型->包装类型, 反之, 拆箱
 2) jdk5 以后(含jdk5) 的自动装箱和拆箱方式
 3) 自动装箱底层调用的是valueOf方法, 比如Integer.valueOf()

手动装箱:

```
//手动装箱
int n1 = 100;
Integer integer = new Integer(n1);
Integer integer1 = Integer.valueOf(n1);
```

手动拆箱:


```
//手动拆箱
//Integer -> int
int i = integer.intValue();
```

● 包装类型和String类型的相互转换 WrapperV

案例演示, 以Integer 和 String 转换为例, 其它类

```
// 包装类型——>String类型
Integer i = 10;
// 方式1:
String s1 = i.toString();
// 方式2:
String s2 = String.valueOf(i);
// 方式3:
String s3 = i + "";
System.out.println(s3);
// String——>包装类
// 方式1:
Integer j = new Integer(s1);
// 方式2:
Integer j2 = Integer.valueOf(s2);
```

Integer 类和 character 类的常用方法:

```
System.out.println(Integer.MIN_VALUE); //返回最小值
System.out.println(Integer.MAX_VALUE); //返回最大值

System.out.println(Character.isDigit('a')); //判断是不是数字
System.out.println(Character.isLetter('a')); //判断是不是字母
System.out.println(Character.isUpperCase('a')); //判断是不是大写
System.out.println(Character.isLowerCase('a')); //判断是不是小写

System.out.println(Character.isWhitespace('a')); //判断是不是空格
System.out.println(Character.toUpperCase('a')); //转成大写
System.out.println(Character.toLowerCase('A')); //转成小写
```

```
public void method1() {
    Integer i = new Integer(1);
    Integer j = new Integer(1);
    System.out.println(i == j); //False
    //所以, 这里主要是看范围 -128 ~ 127 就是直接返回
    Integer m = 1; //底层 Integer.valueOf(1); -> 阅读源码
    Integer n = 1; //底层 Integer.valueOf(1);
    System.out.println(m == n); //T
    //所以, 这里主要是看范围 -128 ~ 127 就是直接返回
    //, 否则, 就new Integer(xx);
    Integer x = 128; //底层 Integer.valueOf(1);
    Integer y = 128; //底层 Integer.valueOf(1);
    System.out.println(x == y); //False
}
```

string 类的理解和创建对象:



String01.java

- 1) String 对象用于保存字符串, 也就是一组字符序列
- 2) 字符串常量对象是用双引号括起的字符序列。例如: "你好"、"12.97"、"boy"等
- 3) 字符串的字符使用Unicode字符编码, 一个字符(不区分字母还是汉字)占两个字节。
- 4) String类较常用构造器(其它看手册):

- String s1 = new String(); //
- String s2 = new String(String original);
- String s3 = new String(char[] a);
- String s4 = new String(char[] a,int startIndex,int count) ;
- 说明:

```
public final class String
    implements Serializable, Comparable<String>, CharSequence {
    // The value is used for character storage
    private final char value[];

    // Create the host code for the string "L"
    private int hash; // Default to 0

    // See serializable() from JDK 1.1.2 for interoperability
    private static final long serialVersionUID = 1234567890123456789L;
}
```

```
//1.String 对象用于保存字符串, 也就是一组字符序列
//2. "jack" 字符串常量, 双引号括起的字符序列
//3. 字符串的字符使用Unicode字符编码, 一个字符(不区分字母还是汉字)占两个字节
//4. String 类有很多构造器, 构造器的重载
// 常用的有 String s1 = new String(); //
//String s2 = new String(String original);
//String s3 = new String(char[] a);
//String s4 = new String(char[] a,int startIndex,int count)
//String s5 = new String(byte[] b)
//5. String 类实现了接口 Serializable【String 可以串行化:可以在网络传输】
//      接口 Comparable【String 对象可以比较大小】
//6. String 是final 类, 不能被其他的类继承
//7. String 有属性 private final char value[]; 用于存放字符串内容
//8. 一定要注意: value 是一个final类型, 不可以修改(需要功力): 即value不能指向
// 新的地址, 但是单个字符内容是可以变化
```

两种创建 String 对象方式的区别:

- 1.直接赋值 String s = "wg";
- 2.调用构造器 String s2 = new String("wg");

1. 方式一: 先从常量池查看是否有"hsp" 数据空间, 如果有, 直接指向; 如果没有则重新创建, 然后指向。s最终指向的是常量池的空间地址
2. 方式二: 先在堆中创建空间, 里面维护了value属性, 指向常量池的hsp空间。如果常量池没有"hsp", 重新创建, 如果有, 直接通过value指向。最终指向的是堆中的空间地址。
3. 画出两种方式的内存分布图

intern

```
public String intern()
```

返回字符串对象的规范实例。

最初为空的字符串池由String类String。

当调用intern方法时，如果池已经包含与equals(Object)方法确定的相当于此String对象的字符串，则返回来自池的字符串。否则，此String对象将添加到池中，并返回对此String对象的引用。

由此可见，对于任何两个字符串s和t， $s.intern() == t.intern()$ 是true当且仅当s.equals(t)是true。

所有文字字符串和字符串值常量表达式都被实体化。字符串文字在The Java™ Language Specification的3.10.5节中定义。

结果

一个字符串与该字符串具有相同的内容，但保证来自一个唯一的字符串池。

```
String a = "hsp"; //a 指向 常量池的 "hsp"
String b = new String("hsp"); //b 指向堆中对象
System.out.println(a.equals(b)); //T
System.out.println(a==b); //F
System.out.println(a==b.intern()); //intern方法自己先查看API
System.out.println(b==b.intern());
```

知识点:

当调用 intern 方法时，如果池已经包含一个等于此 String 对象的字符串（用 equals(Object) 方法确定），则返回池中的字符串。否则，将此 String 对象添加到池中，并返回此 String 对象的引用

解读: (1) b.intern() 方法最终返回的是常量池的地址（对象）。

教育

String类

4) 测试题4 StringExercise04.java 2min

```
String s1 = "hspedu";
String s2 = "java";
String s4 = "java";
String s3 = new String("java");
System.out.println(s2 == s3); //f
System.out.println(s2 == s4); //T
System.out.println(s2.equals(s3)); //T
System.out.println(s1 == s2); //F
```

The diagram illustrates the memory layout for String objects. At the top, a box labeled 'String类' (String class) contains a 'Value' box. Below this, a '池' (pool) box contains two entries: 'hspedu' and 'java'. Arrows indicate the mapping: s1 points to 'hspedu', s2 points to 'java', s4 points to 'java', and s3 points to a new object in memory (not in the pool).


```
String s1 = "hspedu"; //指向常量池" hspedu"
String s2 = "java"; //指向常量池" java"
String s4 = "java"; //指向常量池" java"
String s3 = new String("java"); //指向堆中对象
System.out.println(s2 == s3); // F
System.out.println(s2 == s4); // T
System.out.println(s2.equals(s3)); // T
System.out.println(s1 == s2); // F
```

equals 比较的是内容，==比较的是地址

String类

5) 测试题5 **StringExercise05.java** 5min思考

```
Person p1 = new Person();
p1.name = "hspedu";
Person p2 = new Person();
p2.name = "hspedu";

System.out.println(p1.name.equals(p2.name)); //比较内容: True
System.out.println(p1.name == p2.name); // T
System.out.println(p1.name == "hspedu"); // T
```

String s1 = new String("bcde");
String s2 = new String("bcde");
System.out.println(s1==s2);
//请画出内存布局图

字符串的特性

2) 题2 **StringExercise08.java** 1min思考

```
String a = "hello"; //创建 a对象
String b = "abc"; //创建 b对象
String c=a+b; 创建了几个对象? 画出内存图?
//关键就是要分析 String c = a + b; 到底是如何执行的
//一共有3对象, 如图。


老韩小结: 底层是 StringBuilder sb = new StringBuilder(); sb.append(a); sb.append(b); sb是在堆中, 并且append是在原来字符串的基础上追加的。  
重要规则, String c1 = "ab" + "cd"; 常量相加, 看的是池。 String c1 = a + b; 变量相加, 是在堆中


```

.intern()是指向该变量的内容的常量

String 的常用方法:

String类的常见方法

● String类的常见方法一览[告诉你怎么用，就可以]

- equals
- equalsIgnoreCase
- length
- indexOf
- lastIndexOf
- substring
- trim
- charAt:获取某索引处的字符
- toUpperCase
- toLowerCase
- concat
- compareTo
- toCharArray
- format

● String类的常见方法应用实例1

- equals // 区分大小写，判断内容是否相等
- equalsIgnoreCase // 忽略大小写的判断内容是否相等
- length // 获取字符的个数，字符串的长度
- indexOf // 获取字符在字符串中第1次出现的索引，索引从0开始，如果找不到，返回-1
- lastIndexOf // 获取字符在字符串中最后1次出现的索引，索引从0开始，如找不到，返回-1
- substring // 截取指定范围的子串
- trim // 去前后空格
- charAt:获取某索引处的字符，注意不能使用Str[index] 这种方式。

```
StringExercise09.java x StringExercise10.java x Byte.uml x StringUML.java x StringMethod01.j
// 4.indexOf 获取字符在字符串对象中第一次出现的索引，索引从0开始，如果找不到，
String s1 = "wer@terwe@g";
int index = s1.indexOf('@');
System.out.println(index); // 3
// 5.lastIndexOf 获取字符在字符串中最后一次出现的索引，索引从0开始，如果找不到
s1 = "wer@terwe@g@";
index = s1.lastIndexOf('@');
System.out.println(index); // 11
// 6.substring 截取指定范围的子串
String name = "hello,张三";
// 下面name.substring(6) 从索引6开始截取后面所有的内容
System.out.println(name.substring(6)); // 截取后面的字符
// name.substring(0,5)表示从索引0开始截取，截取到索引 5-1=4位置
System.out.println(name.substring(2,5)); // llo,张
```

```

s1 = s1.concat("林黛玉").concat("薛宝钗").concat("together");
System.out.println(s1); //宝玉林黛玉薛宝钗together
// 4.replace 替换字符串中的字符
s1 = "宝玉 and 林黛玉 林黛玉 林黛玉";
//在s1中, 将 所有的 林黛玉 替换成薛宝钗
// 老韩解读: s1.replace() 方法执行后, 返回的结果才是替换过的。
// 注意对 s1没有任何影响
String s11 = s1.replace("宝玉", "jack");
System.out.println(s1); //jack and 林黛玉 林黛玉 林黛玉

```

StringBuffer 类:

Java.lang.StringBuffer 代表可变的字符序列, 可以对字符串内容进行增删
很多方法和 String 相同, 但 StringBuffer 是可变长度的
StringBuffer 是一个容器

● String VS StringBuffer

- 1) String保存的是字符串常量, 里面的值不能更改, 每次String类的更新实际上就是更改地址, 效率较低 //private final char value[];
- 2) StringBuffer保存的是字符串变量, 里面的值可以更改, 每次StringBuffer的更新实际上可以更新内容, 不用每次更新地址, 效率较高 //char[] value; // 这个放在堆。

● StringBuffer的构造器

构造方法摘要

StringBuffer()

构造一个其中不带字符的字符串缓冲区, 其初始容量为 16 个字符。

StringBuffer(CharSequence seq)

public java.lang.StringBuilder(CharSequence seq) 构造一个字符串缓冲区, 它包含与指定的 CharSequence 相同的字符。

StringBuffer(int capacity) //capacity [容量]

构造一个不带字符, 但具有指定初始容量的字符串缓冲区。即对 char[] 大小进行指定

StringBuffer(String str)

构造一个字符串缓冲区, 并将其内容初始化为指定的字符串内容。

StringBuffer 与 String 的转换

```

//看 String—>StringBuffer
String str = "hello tom";
//方式1 使用构造器
//注意: 返回的才是StringBuffer对象, 对str 本身没有影响
StringBuffer stringBuffer = new StringBuffer(str);
//方式2 使用的是append方法
StringBuffer stringBuffer1 = new StringBuffer();
stringBuffer1 = stringBuffer1.append(str);

```

```
//看看 StringBuffer ->String
StringBuffer stringBuffer3 = new StringBuffer("韩顺平教育");
//方式1 使用StringBuffer提供的 toString方法
String s = stringBuffer3.toString();
//方式2: 使用构造器来搞定
String s1 = new String(stringBuffer3);
```

StringBuffer 类的常见方法:

- 1) 增 append
- 2) 删 delete(start, end) (左闭右开区间)
- 3) 改 replace(start, end, string)//将 start--end 间的内容替换掉, 不含 end
- 4) 查 indexOf//查找子串在字符串第一次出现的索引, 如果找不到就返回-1
- 5) 插 insert
- 6) 获取长度 length

```
//new Scanner(System.in)
String price = "123564.59";
StringBuffer sb = new StringBuffer(price);
//先完成一个最简单的实现123,564.59
//找到小数点的索引, 然后在该位置的前3位, 插入, 即可
//int i = sb.lastIndexOf(".");

//上面的两步需要做一个循环处理, 才是正确的
for( int i = sb.lastIndexOf("."); i > 0 ; i -=3) {
    sb = sb.insert(i - 3, ",");
}
System.out.println(sb);//123,564.59
```

```
//new Scanner(System.in)
String price = "8123564.59";
StringBuffer sb = new StringBuffer(price);
//先完成一个最简单的实现123,564.59
//找到小数点的索引, 然后在该位置的前3位, 插入, 即可
int i = sb.lastIndexOf(".");
sb = sb.insert(i - 3, ",");

//上面的两步需要做一个循环处理, 才是正确的
for( int i = sb.lastIndexOf(".") - 3; i > 0 ; i -=3) {
    sb = sb.insert(i , ",");
}
System.out.println(sb);//8,123,564.59
```

StringBuilder 类:

StringBuilder01.java

- 1) 一个可变的字符序列。此类提供一个与 `StringBuffer` 兼容的 API，但不保证同步(`StringBuilder` 不是线程安全)。该类被设计用作 `StringBuffer` 的一个简易替换，用在字符串缓冲区被单个线程使用的时候。如果可能，建议优先采用该类，因为在大多数实现中，它比 `StringBuffer` 要快 [后面测]。
- 2) 在 `StringBuilder` 上的主要操作是 `append` 和 `insert` 方法，可重载这些方法，以接受任意类型的数据。

`StringBuilder` 大多用于单线程而 `StringBuffer` 多用于多线程

教育

StringBuilder类

- String、StringBuffer 和StringBuilder的比较
- 1) `StringBuilder` 和 `StringBuffer` 非常类似，均代表可变的字符序列，而且方法也一样
- 2) `String`：不可变字符序列，效率低，但是复用率高。
- 3) `StringBuffer`：可变字符序列、效率较高(增删)、线程安全
- 4) `StringBuilder`：可变字符序列、效率最高、线程不安全
- 5) `String`使用注意说明：
string s="a"; //创建了一个字符串
s += "b"; //实际上原来的"a"字符串对象已经丢弃了，现在又产生了一个字符串s+"b" (也就是"ab")。如果多次执行这些改变串内容的操作，会导致大量副本字符串对象存留在内存中，降低效率。如果这样的操作放到循环中，会极大影响程序的性能 => 结论：如果我们对String 做大量修改，不要使用String

```
public StringBuilder append(StringBuffer sb) {
    super.append(sb);
    return this;
}

@Override
public StringBuilder append(CharSequence s) {
    super.append(s);
    return this;
}
```

没有线程安全控制

```
@Override
public synchronized StringBuffer append(Object obj) {
    toStringCache = null;
    super.append(String.valueOf(obj));
    return this;
}

@Override
public synchronized StringBuffer append(String str) {
    toStringCache = null;
    super.append(str);
    return this;
}
```

● String、StringBuffer 和StringBuilder的选择

使用的原则, 结论:

1. 如果字符串存在大量的修改操作，一般使用 `StringBuffer` 或 `StringBuilder`
2. 如果字符串存在大量的修改操作，并在单线程的情况，使用 `StringBuilder`
3. 如果字符串存在大量的修改操作，并在多线程的情况，使用 `StringBuffer`
4. 如果我们字符串很少修改，被多个对象引用，使用 `String`，比如配置信息等

`StringBuilder` 的方法使用和 `StringBuffer` 一样，不再说。

`Math` 类：包括用于执行基本数学运算的方法，如初等指数、对数、平方根和三角函数

MathMethod.java

我们演示下Math 类常见的方法，看老师演示。

- 1) abs 绝对值
- 2) pow 求幂
- 3) ceil 向上取整
- 4) floor 向下取整
- 5) round 四舍五入
- 6) sqrt 求开方
- 7) random 求随机数 //思考:
请写出获取 a-b之间的一个随机整数,a,b均为整数? 2-7
- 8) max 求两个数的最大值
- 9) min 求两个数的最小值

练习题：获取一个 a-b 之间的一个随机整数。

```
//1.abs 绝对值
int abs = Math.abs(9);
System.out.println(abs);
//2.pow 求幂
double pow = Math.pow(-3.5, 4);
System.out.println(pow);
//3.ceil 向上取整,返回>=该参数的最小整数:
double ceil = Math.ceil(-3.0001);
System.out.println(ceil);
//4.floor 向下取整, 返回<=该参数的最大整数
double floor = Math.floor(-4.999);
System.out.println(floor);
//5.round 四舍五入 Math.floor(该参数+0.5)
long round = Math.round(-5.001);
System.out.println(round);
//6.sqrt 求开方
double sqrt = Math.sqrt(-9.0);
System.out.println(sqrt);
//7.random 返回随机数 [0—1)
//[a-b]:int num = (int)(Math.random()* (b-a+1) + a)
double random = Math.random();
System.out.println(random);
```

```
// 老韩解读 Math.random() * (b-a) 返回的就是 0 <= 数 <= b-a
// (1) (int)(a) <= x <= (int)(a + Math.random() * (b-a + 1) )
// (2) 使用具体的数给小伙伴介绍 a = 2 b = 7
// (int)(a + Math.random() * (b-a + 1) ) = (int)( 2 + Math.random()*6)
// Math.random()*6 返回的是 0 <= x < 6 小数
// 2 + Math.random()*6 返回的就是 2<= x < 8 小数
// (int)(2 + Math.random()*6) = 2 <= x <= 7
// (3) 公式就是 (int)(a + Math.random() * (b-a + 1) )
for(int i = 0; i < 100; i++) {
    System.out.println((int)(2 + Math.random() * (7 - 2 + 1)));
}
```

Arrays 类:

Arrays类常见方法应用案例

ArraysMethod01.java

Arrays里面包含了一系列静态方法，用于管理或操作数组(比如排序和搜索)。

- 1) toString 返回数组的字符串形式

Arrays.toString(arr)

- 2) sort 排序 (自然排序和定制排序) Integer arr[] = {1, -1, 7, 0, 89};

- 3) binarySearch 通过二分搜索法进行查找，要求必须排好序

```
int index = Arrays.binarySearch(arr, 3);
```

Arrays 类常见方法

```
// 老韩解读
//1. 使用 binarySearch 二叉查找
//2. 要求该数组是有序的，如果该数组是无序的，不能使用binarySearch
//3. 如果数组中不存在该元素，就返回 return -(low + 1); // key not found
int index = Arrays.binarySearch(arr, 568);
System.out.println("index=" + index);
```

ArraysMethod02.java

4) copyOf 数组元素的复制

```
Integer[] newArr = Arrays.copyOf(arr, arr.length);
```

5) fill 数组元素的填充

```
Integer[] num = new Integer[]{9,3,2};
```

```
Arrays.fill(num, 99);
```

6) equals 比较两个数组元素内容是否完全一致

```
boolean equals = Arrays.equals(arr, arr2);
```

7) asList 将一组值, 转换成list

```
List<Integer> asList = Arrays.asList(2,3,4,5,6,1);
```

```
System.out.println("asList=" + asList);
```

```
//copyOf 数组元素的复制
// 老韩解读
//1. 从 arr 数组中, 拷贝 arr.length个元素到 newArr数组中
//2. 如果拷贝的长度 > arr.length 就在新数组的后面 增加 null
//3. 如果拷贝长度 < 0 就抛出异常NegativeArraySizeException
Integer[] newArr = Arrays.copyOf(arr, -1);
System.out.println("==拷贝执行完毕后==");
System.out.println(Arrays.toString(newArr));
```

```
//fill 数组元素的填充
Integer[] num = new Integer[]{9,3,2};
//老韩解读
//1. 使用 99 去填充 num数组, 可以理解成是替换原理的元素
Arrays.fill(num, 99);
System.out.println("==num数组填充后==");
System.out.println(Arrays.toString(num));
```

```
//equals 比较两个数组元素内容是否完全一致
Integer[] arr2 = {1, 2, 90, 123};
//老韩解读
//1. 如果arr 和 arr2 数组的元素一样, 则方法true;
//2. 如果不是完全一样, 就返回 false
boolean equals = Arrays.equals(arr, arr2);
System.out.println("equals=" + equals);
```

System 类常见方法

字母	日期或时间元素	表示	示例
G	Era 标志符	Text	AD
y	年	Year	1996; 96
M	年中的月份	Month	July; Jul; 07
w	年中的周数	Number	27
W	月份中的周数	Number	2
D	年中的天数	Number	189
d	月份中的天数	Number	10
F	月份中的星期	Number	2
E	星期中的天数	Text	Tuesday; Tue
a	Am/pm 标记	Text	PM
H	一天中的小时数 (0-23)	Number	0
k	一天中的小时数 (1-24)	Number	24
K	am/pm 中的小时数 (0-11)	Number	0
h	am/pm 中的小时数 (1-12)	Number	12
m	小时中的分钟数	Number	30
s	分钟中的秒数	Number	55
S	毫秒数	Number	978
z	时区	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	时区	RFC 822 time zone	-0800

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy年MM月dd日 hh:mm:ss E");
String format = sdf.format(d1); // format:将日期转换成指定格式的字符串
System.out.println("当前日期=" + format);

//老韩解读
//1. 可以把一个格式化的String 转成对应的 Date
//2. 得到Date 仍然在输出时, 还是按照国外的形式, 如果希望指定格式输出, 需要
String s = "1996年01月01日 10:20:30 星期一";
Date parse = sdf.parse(s);
System.out.println("parse=" + sdf.format(parse));
```

● 第二代日期类

1) 第二代日期类, 主要就是 Calendar类(日历)。

```
public abstract class Calendar extends Object implements Serializable,
Cloneable, Comparable<Calendar>
```

2) Calendar 类是一个抽象类, 它为特定瞬间与一组诸如 YEAR、MONTH、DAY_OF_MONTH、HOUR 等 日历字段之间的转换提供了一些方法, 并为操作日历字段 (例如获得下星期的日期) 提供了一些方法。

● 第三代日期类

➢ 前面两代日期类的不足分析

JDK 1.0中包含了一个java.util.Date类，但是它的大多数方法已经在JDK 1.1引入Calendar类之后被弃用了。而Calendar也存在问题：

- 1) 可变性：像日期和时间这样的类应该是不可变的。
- 2) 偏移性：Date中的年份是从1900开始的，而月份都从0开始。
- 3) 格式化：格式化只对Date有用，Calendar则不行。
- 4) 此外，它们也不是线程安全的；不能处理闰秒等（每隔2天，多出1s）。

➢ 第三代日期类常见方法

1) LocalDate(日期)、LocalTime(时间)、LocalDateTime(日期时间) JDK8

LocalDate只包含日期，可以获取日期字段

• LocalTime只包含时间，可以获取时间字段

LocalDateTime包含日期+时间，可以获取日期和时间字段

案例演示[后ppt]:

```
LocalDateTime ldt = LocalDateTime.now(); //LocalDate.now();//LocalTime.now()
System.out.println(ldt);
ldt.getYear();ldt.getMonthValue();ldt.getMonth();ldt.getDayOfMonth();
ldt.getHour();ldt.getMinute();ldt.getSecond();
```

3) Instant 时间戳

类似于Date

提供了一系列和Date类转换的方式

Instant——>Date:

Date date = Date.from(instant);

Date——>Instant:

Instant instant = date.toInstant();

案例演示:

```
Instant now = Instant.now();
```

```
System.out.println(now);
```

```
Date date = Date.from(now);
```

```
Instant instant = date.toInstant();
```

4) 第三代日期类更多方法

- LocalDateTime类
- MonthDay类:检查重复事件
- 是否是闰年
- 增加日期的某个部分
- 使用plus方法测试增加时间的某个部分
- 使用minus方法测试查看一年前和一年后的日期
- 其他的方法，老师就不说，使用的时候，自己查看API使用即可