# JavaSE02-面向对象高级

# final 关键字

## final关键字的作用

（1） 采用 final 修饰的类不能被继承

（2） 采用 final 修饰的方法不能被覆盖

（3） 采用 final 修饰的变量不能被修改

（4） final 修饰的变量必须显示初始化

（5） 如果修饰的引用，那么这个引用只能指向一个对象，也就是说这个引用不能再次赋值，但被指向的对象属性是可以修改的

（6） 构造方法不能被 final 修饰

## final关键字的使用

案例1：

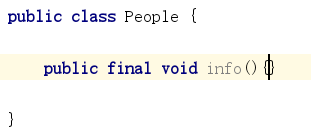
采用 final 修饰的类不能被继承

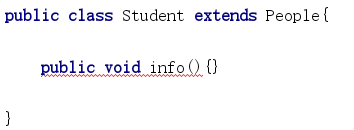




案例2：

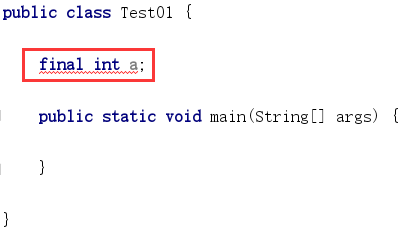
采用 final 修饰的方法不能被覆盖





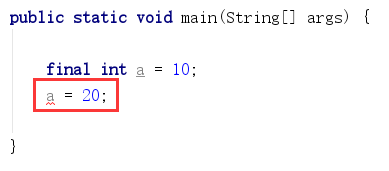
案例3：

final 修饰的成员变量必须初始化



案例4：

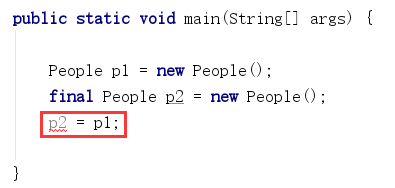
采用 final 修饰的变量(基本类型)不能被修改



案例5：

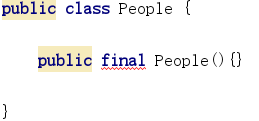
如果修饰的引用，那么这个引用只能指向一个对象，也就是说这个

引用不能再次赋值，但被指向的对象的属性值是可以修改的



案例6：

final不能修饰构造方法



# 抽象类

## 抽象类概述

抽象表示不具体的意思，所谓抽象类就是相对不具体的类。

例如我们有Person、Student 和 Employee这3个类，从我们使用的角度来看主要对 Student类 和Employee类 进行实例化，Person 中主要包含了一些公共的属性和方法，所以作为父类。而 Person 我们通常不会实例化，所以我们可以把它定义成抽象的：

抽象类具有如下特征：

（1）在 java 中采用 abstract 关键字定义的类就是抽象类，采用 abstract 关键字定义的方法就是抽象方法

（2） 抽象的方法只需在抽象类中提供声明，不需要实现（没有方法体）

（3） 如果一个类中含有抽象方法，那么这个类必须定义成抽象类

（4） 如果这个类是抽象的，那么这个类被子类继承，抽象方法必须被重写。如果在子类中不复写该抽象方法，那么子类也必须是抽象类。 （重写，覆写，覆盖，复写 都是一回事）

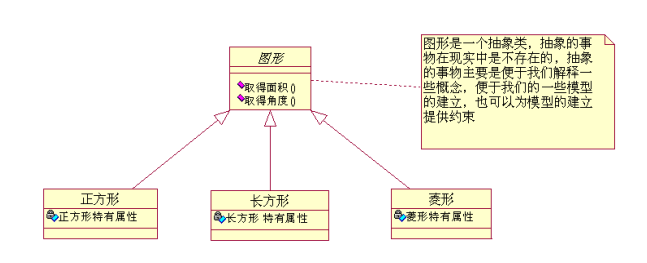
（5） 抽象类是不能实例化的

（6） 抽象类不能被 final 修饰

（7） 抽象方法不能被 final 修饰，因为抽象方法就是被子类实现的

抽象类中可以包含非抽象方法以及方法实现，可以将一些公共的代码放到抽象类中。

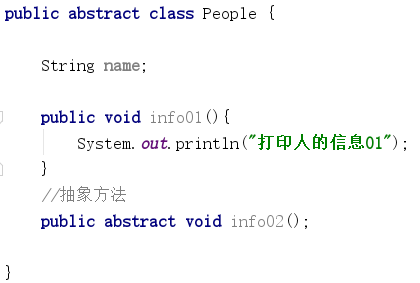
如图所示：



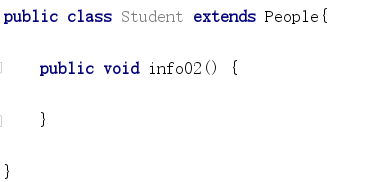
## 抽象类的使用

案例1：

创建抽象类

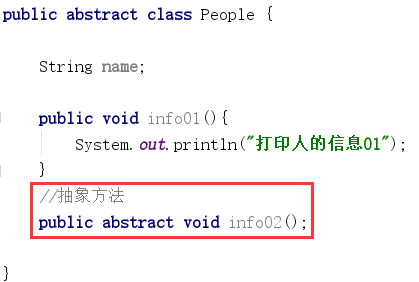


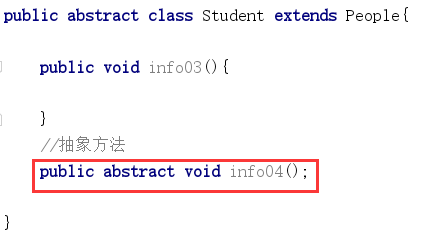
子类继承抽象父类后，必须覆盖抽象方法，否则编译报错

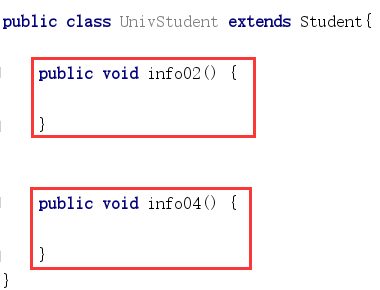


案例2：

子类也是抽象类，则不需要覆盖父类的抽象方法，但是最终继承抽象类的非抽象类必须重写祖辈所有的抽象方法







案例3：

不能使用final修饰抽象类，这是矛盾的



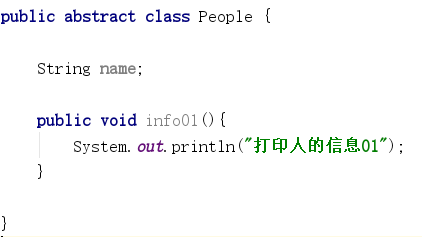
案例4：

不能使用final修饰抽象方法，这也是矛盾的



案例5：

抽象类中可以没有抽象方法



案例6：

抽象类不能被实例化



## 访问控制权限

java 访问级别修饰符主要包括：private protected 和 public，可以限定其他类对该类、属性和方法的使用权限。 其他

Y：Yes 能访问到

N：No 访问不到



注意：

以上对类的修饰只有：public 和 default，内部类除外（内部类还没有学习）

# 接口

## 接口概述

接口我们可以看作是抽象类的一种特殊情况，他比抽象类更加的抽象。

使用接口有如下的注意事项：

（1）在 java 中接口采用 interface 声明

（2）在接口中只能定义抽象的方法和常量

（3）接口中的方法默认都是 public abstract 的，不能更改

（4） 接口中的变量默认都是 public static final 类型的，不能更改，所以必须显示的初始化

（5）接口不能被实例化，接口中没有构造函数的概念

（6）接口之间可以继承

（7）我们的类是通过 implements 关键字来实现接口的

（8）如果一个类实现了接口，那么接口中所有的方法必须实现

（9）一个类可以同时实现多个接口

（10）类在实现接口的同时还可以继承父类

## 接口的使用

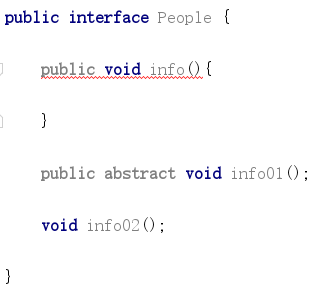
案例1：

建立接口的语法：



案例2：

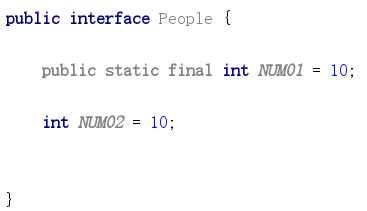
接口中的方法默认都是 public abstract 的，不能更改



如以上案例，其中info02方法没有写public abstract，但是在接口中默认也会加上public abstract。

案例3：

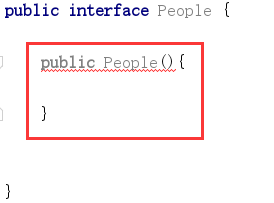
接口中的变量默认都是 public static final 类型的，不能更改，所以必须显示的初始化



如以上案例，NUM02变量我们没有加public static final，但是在接口中也会默认加上public static final。另外值得注意的是常量的标识符我们习惯约定成所有字母大写，这点要求大家记住。

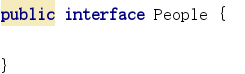
案例4：

接口不能被实例化，接口中没有构造函数的概念



案例5：

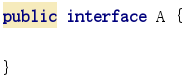
接口之间可以继承

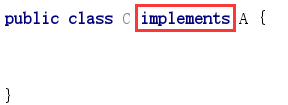




案例6：

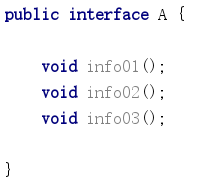
我们的类是通过 implements 关键字来实现接口的

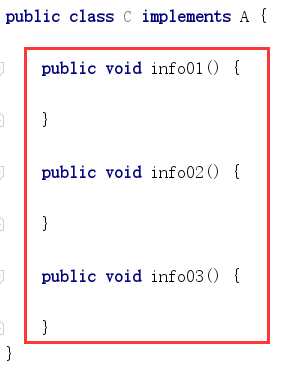




案例7：

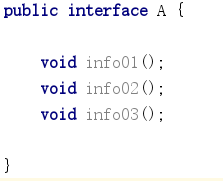
如果一个类实现了接口，那么接口中所有的方法必须实现

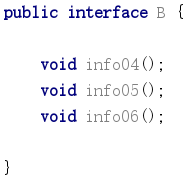


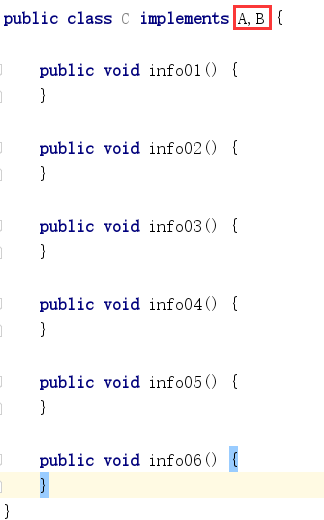


案例8：

一个类可以同时实现多个接口，实现的接口之间使用逗号分隔开即可。实现了接口之后，要覆盖所有实现的接口中的抽象方法。 这样做无形之间相当于实现了多继承。







案例9：

类在实现接口的同时还可以继承父类

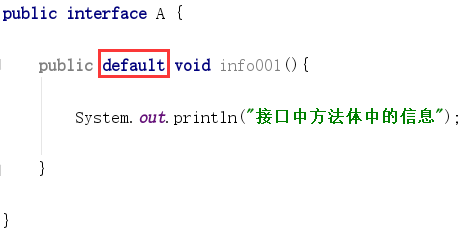
使用时先继承父类，再实现接口





案例10：

接口中也可以有非抽象方法的实现，需要再方法中加入default修饰符。但是这样用法以后基本不用，做个了解即可。



案例11：

使用多态



这一点与抽象类的作用是一致的

## 接口应用扩展

在 java 中接口其实描述了类需要做的事情，类要遵循接口的定义来做事，使用接口到底有什么本质的好处？

可以归纳为三点：

（1） 采用接口明确的声明了它所能提供的**服务**，这就相当于是为方法制定了**标准**，实现类只要实现相应的标准就好了。

实现了可接插性。

（2） 解决了 Java 单继承的问题 ，可以同时实现多个接口

（3） 使用多态让我们的代码变得更加灵活

接口（抽象类）多态性的总结：

在学习多态之前，我们都是依赖具体的实现

例如：Dog d = new Dog();

在学习多态之后，我们应该依赖抽象类和接口

例如：Animal a = new Dog();

而依赖抽象类和接口，就像你的手机电池一样：你的手机只依赖电池（电池是一个抽象的事物）， 而不依赖某个厂家的电池(某个厂家的电池就是具体的事物了) 。

因为你依赖了抽象的事物，每个抽象的事物都有不同的实现 ，这样你就可以利用多态的机制完成动态绑定，进行互换，是程序具有较高的灵活。

未来的实际项目开发，我们尽量遵循面向接口（抽象）编程，而不要面向实现编程。

## 接口和抽象类的区别

相同点：  
（1）接口和抽象类都不能被实例化。  
（2）接口的实现类和抽象类的（非抽象）子类，必须覆盖接口或者抽象类中的抽象方法。

不同点：  
（1）接口只能定义抽象方法不能实现方法（JDK8之后提供了default方法，但是应用不多），抽象类既可以定义抽象方法，也可以实现方法。  
（2）接口可以同时实现多个，但是只能继承一个父类。  
（3）接口强调的是功能，抽象类强调的是所属关系。  
（4）接口中的所有成员变量 为public static final， 静态不可修改，当然必须初始化。接口中的所有方法都是public abstract 公开抽象的。而且不能有构造方法。抽象类就比较自由了，和普通的类差不多，可以有抽象方法也可以没有，可以有正常的方法，也可以没有。

总结：

在实际项目开发中，优先选择接口（因为继承抽象类后，此类将无法再继承，所以会丧失此类的灵活性）

## 本章习题

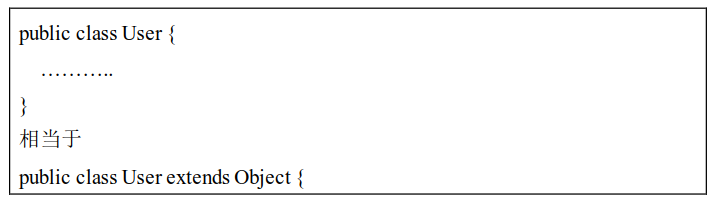
看课件

# Object类

## Object类概述

Object 类是所有 Java 类的根基类，如果在类的声明中未使用 extends 关键字指明其基类，则默认基类为 Object 类。

例如：



## toString方法

toString方法为我们返回该对象的字符串表示。

通常 toString 方法会返回一个“以文本方式表示”此对象的字符串，Object 类的 toString 方法返回一个字符串，该字符串由类名加标记@和此对象哈希码的无符号十六进制表示组成，Object 类 toString 源代码如下：

getClass().getName() + '@' + Integer.toHexString(hashCode())

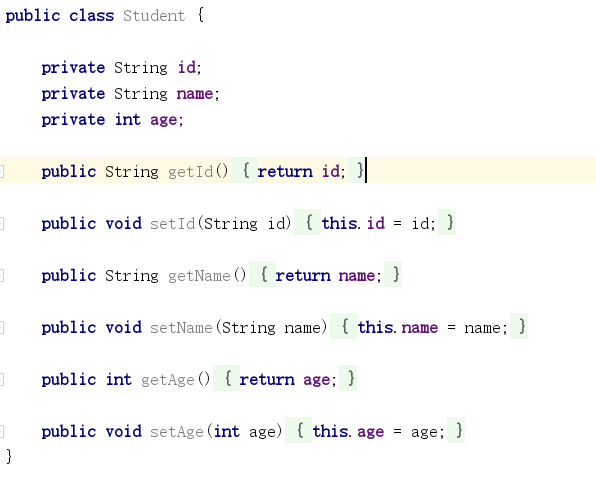
在进行 String 与其它类型数据的连接操作时，如：

System.out.println(student);

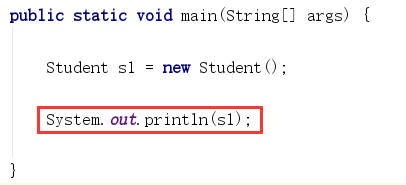
自动调用该对象的 toString()方法

例如：

我们现在有一个标准的学生实体类



我们直接打印学生的对象



执行结果：



根据输出结果，表明了Student类默认继承了Object类，在输出对象s的时候其实就是输出了s.toString();

也就是说：

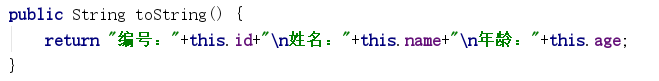


执行结果是一样的：

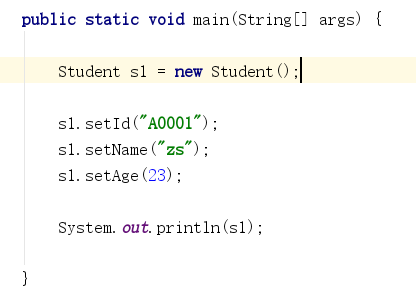


我们可以将以上输入的结果看做是对象的地址，我们在使用系统的过程中一般不会去观察地址。所以将来为了更方便的使用该方法，我们都是将toString方法进行覆盖，然后打印我们想要的结果。

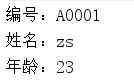
例如我们可以将toString方法覆盖具有如下功能



测试



执行结果：



## finalize方法

垃圾回收器（**G**arbage **C**ollection），**也叫 GC**，垃圾回收器主要有以下特点：

（1） 当对象不再被程序使用时，垃圾回收器将会将其回收

（2） 垃圾回收是在后台运行的，我们无法命令垃圾回收器马上回收资源，但是我们可以告诉他，尽快回收资源（System.gc 和 Runtime.getRuntime().gc()）

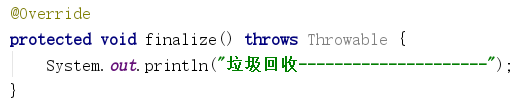
（3） 垃圾回收器在回收某个对象的时候，首先会调用该对象的 finalize 方法

（4） GC 主要针对堆内存

当垃圾收集器将要收集某个垃圾对象时将会调用 finalize，建议不要使用此方法，因为此方法的运行时间不确定，如果执行此方法出现错误，程序不会报告，仍然继续运行。

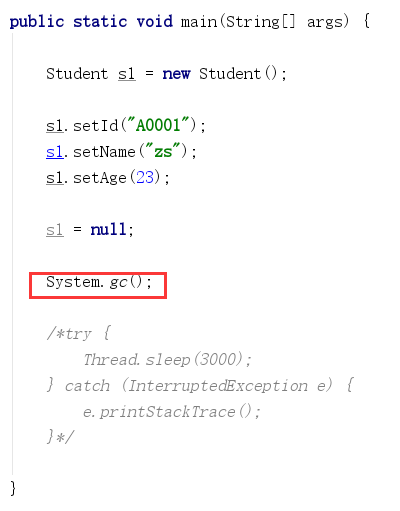
在Student类中覆盖finalize方法

这个方法在垃圾回收时会自动调用



测试：

System.gc()这个方法是为了”提示”垃圾回收器要回收垃圾了



执行结果：

系统”会为我们打印”

**（注意：小概率不打印信息，因为有可能没有及时回收）**



之所以会有这种结果，是因为：

首先我们将对象s设置为 null ，s 不再执行堆中的对象 ，那么此时堆中的对象就是垃圾对象 。垃圾收集（GC）就会收集此对象 。

但是GC 不会马上收集，收集时间不确定。

但是我们又执行了System.*gc*();可以告诉 GC，马上来收集垃圾。但GC也不确定会马上来。

总结：

在实际项目开发中，以上这种写法没有多大的意义， 执行完成方法，所有的局部变量的生命周期全部结束。

所以堆区中的对象就变成垃圾了（因为没有引用指向对象了）

所以不用我们手动进行s = null的操作。

最后java中的垃圾自动回收机制就会在”适当的时机”将堆中的垃圾回收掉。

## ==与 equals 方法

### ==的使用

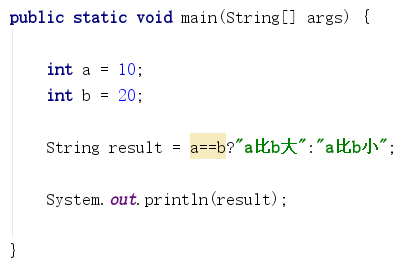
等号“==”做比较使用，返回boolean类型值。

（1）等号“==”在基本数据类型的使用中，比较的是值。

（2）等号“==”在引用数据类型的使用中，比较的是内存地址。

案例1：

比较基本数据类型int

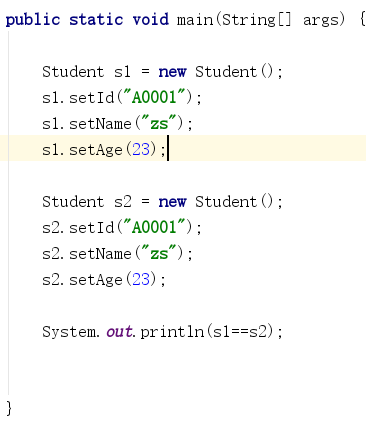


执行结果：



案例2：

比较引用数据类型



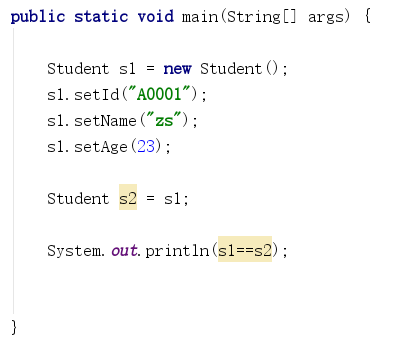
虽然对象中的属性值相同，但是由于比较的是内存地址，

所以执行结果是：



案例3：

让两个引用类型变量，指向同一块地址

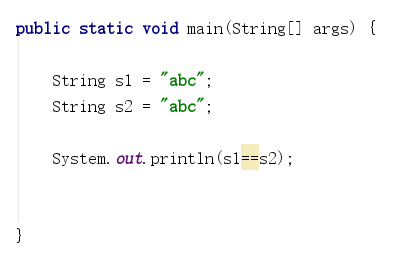


执行结果：



案例4：

比较字符串



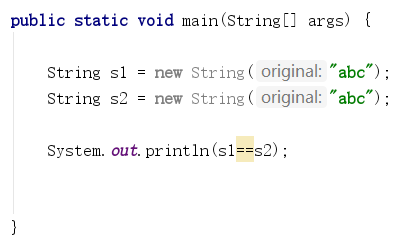
执行结果



字符串是引用数据类型，那么为什么与比较学生对象的结果不一样呢，那是因为字符串属于一种特殊的引用数据类型。

由于在实际项目开发中我们会大量的使用字符串，所以比较占用内存。字符串是经过内存优化的一种存储方式，如果以前在内存中有字符串abc了，下面如果再创建一个引用，则默认指向的就是已再内存中存在的abc。

但是如果我们是以new的形式来创建字符串，则结果就不同了：



执行结果：



### equals方法的使用

equals方法用来比较字符串的值是否相同，未来比较字符串主要使用的就是equals方法。

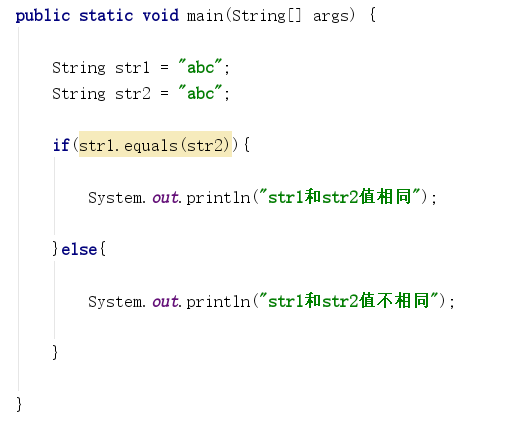
语法：

值1.equals(值2)

返回boolean类型

如果两个值相同则返回true，否则返回false

案例：





注意：

使用equals方法比较的就是单纯的值，与地址的概念没有任何关系。

练习：测试以上案例

# 内部类

内部类在实际开发中应用非常少，本章内容了解即可。在一个类的内部定义的类，称为内部类 。

内部类主要分类：

（1） 实例内部类

（2） 局部内部类

（3） 静态内部类

（4）如果按照特殊用法，还能扩展出来一个匿名内部类

## 实例内部类

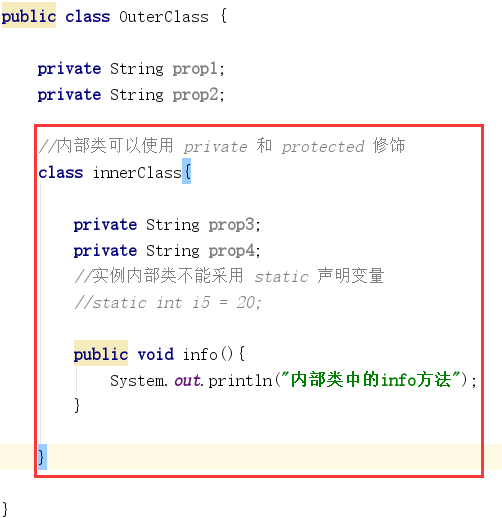
在类体中创建一个内部类

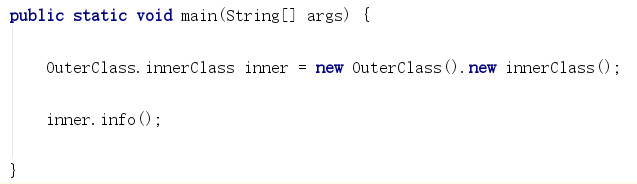
（1） 创建实例内部类，外部类的实例必须已经创建

（2） 实例内部类会持有外部类的引用

（3） 实例内部不能定义 static 成员，只能定义实例成员

案例：



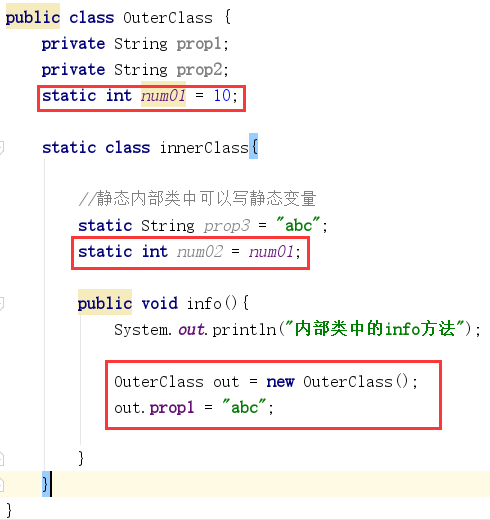


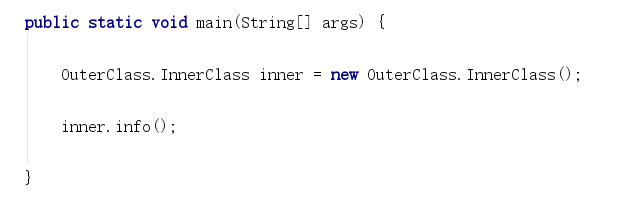
## 静态内部类

使用static修饰的内部类，为静态内部类

（1） 静态内部类不会持有外部的类的引用，创建时可以不用创建外部类

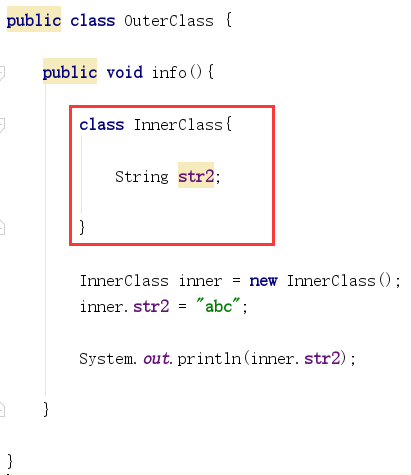
（2） 静态内部类可以使用外部的静态变量，如果访问外部类的成员变量必须通过外部类的实例访问

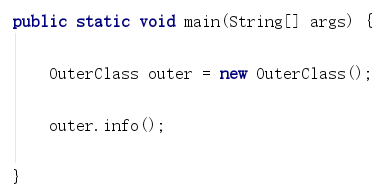




## 局部内部类

局部内部类是在方法中定义的，它只能在当前方法中使用。和局部变量的作用一样。局部内部类和实例内部类一致，不能包含静态成员。

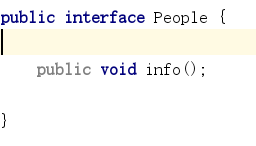


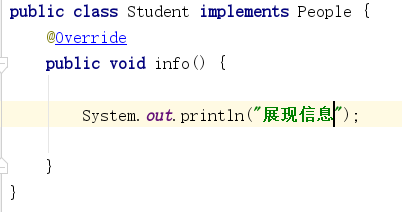


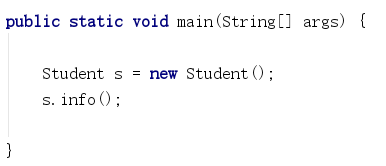
## 匿名内部类

匿名内部类是一种特殊的内部类，该类没有名字。

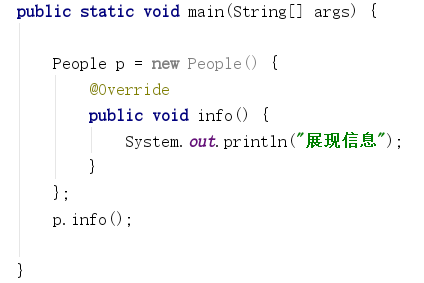
在没有使用匿名内部类的时候，我们可以这么来使用接口的方法







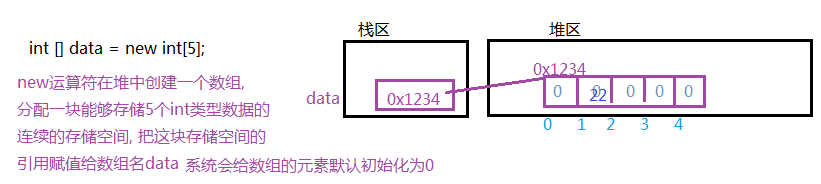
但是如果现在有一个需求是，只是需要定义一下接口的方法，在程序运行过程中，只使用一次。那么我们为了只使用一次的接口方法就创建了一个实现类，那么显然是不恰当的做法。这时我们就可以使用匿名内部类。



# 数组高级

## 数组内存分析

创建数组的内在分析:



## 可变长参数（了解）

可变长参数用于接收任意多个数据.

定义格式如下:

方法名 ( 数据类型 参数名, 数据类型 ... 可变长参数名 ){}

说明:

1)在参数类型与参数名中间使用 ... 三个小点表示它是可变长参数

2)可变长参数可以接收任意个数据

3)一个方法最多只能 有一个变长参数

4)变长参数只能放在参数列表的最后

5)在方法体中,可以简单的把变长参数当作数组使用

如定义方法,计算任意个整数的和:

public static void sum( int ... data){}

## 数组扩容

定义了数组之后 ,在数组中存储元素的个数就确定了. 如果想要存储更多的数据,就需要对数组进行扩容.数组扩容其实就是定义一个更大的数组, 把原来数组中的元素复制到新数组中, 让原来的数组名指向新的数组.

## 数组特点及常用方式

数组是内存中一块连续的存储空间.通过下标（索引）可以快速访问数组的元素,不管数组的长度有多大,可以通过下标快速计算出每个元素的存储地址,访问速度快. 数组的缺点是插入/删除元素效率低. 在插入元素时,可能需要数组扩容, 需要复制/移动大量的元素.所以数组适合应用于以访问为主,很少进行添加/删除操作的场景.

**总结：**

**查询快（连续的内存空间，而且有索引）**

**添加删除慢**

## 数组相关算法

### 冒泡排序

冒泡由小到大排序思路:从前向后进行两两比较,如果前面的数大于(小于)后面的数就交换.

### 选择排序

选择由小到大排序思路:从当前的数中选择最小(最大)的交换到前面

### 二分查找：是要查找指定元素的下标

二分查找其目的是要找到指定元素的下标。

二分查找的前提是数组已经由小到大排序完毕了.二分查找思路是始终与中间的元素比较大小,如果要查找的元素小于中间数,查找范围缩小到左一半;如果要查找的元素比中间的元素大,把查找范围缩小到右一半.

## 对象数组

对象数组就是数组中存储的是对象.实际上数组元素类型是引用类型的数组都可以称为对象数组.对象数组元素其实存储的是对象的引用.

练习1：

（1）定义一个Student类,定义数组存储10个Student对象

（2）判断数组中有没有叫zs的学生

## Arrays工具类

java.util.Arrays类中定义一组操作数组的方法,如:

|  |  |
| --- | --- |
| static [List](mk:@MSITStore:D:\yexiao1793\JDK8.0API.chm::/java/util/../../java/util/List.html)<T> | [asList](mk:@MSITStore:D:\yexiao1793\JDK8.0API.chm::/java/util/../../java/util/Arrays.html#asList-T...-)(T... a) 把数组转换为List列表 |
| static int | [binarySearch](mk:@MSITStore:D:\yexiao1793\JDK8.0API.chm::/java/util/../../java/util/Arrays.html#binarySearch-int:A-int-)(int[] a, int key)在a数组中采用二分查找,返回key元素的索引值 |
| static int[] | [copyOf](mk:@MSITStore:D:\yexiao1793\JDK8.0API.chm::/java/util/../../java/util/Arrays.html#copyOf-int:A-int-)(int[] original, int newLength) 复制original数组,新的数组长度是newLength |
| static int[] | [copyOfRange](mk:@MSITStore:D:\yexiao1793\JDK8.0API.chm::/java/util/../../java/util/Arrays.html#copyOfRange-int:A-int-int-)(int[] original, int from, int to) 把original数组中[from, to)范围内的元素复制到新数组中 |
| static void | [fill](mk:@MSITStore:D:\yexiao1793\JDK8.0API.chm::/java/util/../../java/util/Arrays.html#fill-int:A-int-int-int-)(int[] a, int fromIndex, int toIndex, int val) 把 a数组中[from,to)范围的元素使用val填充 |
| static void | [parallelSort](mk:@MSITStore:D:\yexiao1793\JDK8.0API.chm::/java/util/../../java/util/Arrays.html#parallelSort-int:A-)(int[] a) 对a数组中元素并行 排序,适合于元素非常多的情况 |
| static void | [sort](mk:@MSITStore:D:\yexiao1793\JDK8.0API.chm::/java/util/../../java/util/Arrays.html#sort-int:A-)(int[] a) 排序 |
| static <T> void | [sort](mk:@MSITStore:D:\yexiao1793\JDK8.0API.chm::/java/util/../../java/util/Arrays.html#sort-T:A-int-int-java.util.Comparator-)(T[] a, int fromIndex, int toIndex, [Comparator](mk:@MSITStore:D:\yexiao1793\JDK8.0API.chm::/java/util/../../java/util/Comparator.html)<? super T> c) 对a数组[from,to)范围的元素进行排序,根据comparator比较器比较大小 |
| static [String](mk:@MSITStore:D:\yexiao1793\JDK8.0API.chm::/java/util/../../java/lang/String.html) | [toString](mk:@MSITStore:D:\yexiao1793\JDK8.0API.chm::/java/util/../../java/util/Arrays.html#toString-int:A-)(int[] a) 可以把数组中元素转换为字符串 |
| static [String](mk:@MSITStore:D:\yexiao1793\JDK8.0API.chm::/java/util/../../java/lang/String.html) | [deepToString](mk:@MSITStore:D:\yexiao1793\JDK8.0API.chm::/java/util/../../java/util/Arrays.html#deepToString-java.lang.Object:A-)([Object](mk:@MSITStore:D:\yexiao1793\JDK8.0API.chm::/java/util/../../java/lang/Object.html)[] a) 把多维数组转换为字符串 |

# 常用类

## 字符串相关的常用类

字符串就是一串字符序列.

Java提供了String,StringBuffer,StringBuilder等类来封装字符串,并提供了一系列方法来操作字符串.

String字符串是不可变的,即一旦创建String对象后,包含在这个对象中的字符序列是不可改变的,直到这个对象被销毁.

StringBuffer/StringBuilder字符串对象是可变的,当创建StringBuilder对象后,通过append(),reverse()等方法可以改变这个字符串对象中的字符序列. 最后调用StringBuilder对象的toString()方法最终转换为我们熟悉的String对象.

### String

#### 创建String对象

可以直接给String类型的变量赋值字符串字面量,也可以通过String构造方法创建String对象.String类的常用构造方法有:

String() 无参构造,创建一个空串

String(byte[] bytes) 以当前默认编码把指定的字节数组转换为字符串

String(byte[] bytes, Charset charset) 把bytes字节数组以指定的编码charset转换为字符串

String(byte[] bytes, int offset, int length) 把bytes字节数组中从offset开始的length个字节转换为字符串

String(byte[] bytes, String charsetName) 把bytes字节数组以指定的编码charsetName转换为字符串

String(char[] value) 把字符数组value中的所有字符都转换为字符串

String(char[] value, int offset, int count) 把字符数组value中从offset开始的count个字符转换为字符串

String(String original) 根据现在的original字符串创建一个新的字符串

#### String类的常用方法

|  |  |
| --- | --- |
| char | [charAt](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#charAt-int-)(int index) 返回当前字符串中index位置的字符. |
| **int** | [**compareTo**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#compareTo-java.lang.String-)**(**[**String**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html)**anotherString) 比较当前字符串与参数字符串的大小. 如果第一个字符串大返回正数, 参数字符串大返回负数,一样返回0** |
| int | [compareToIgnoreCase](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#compareToIgnoreCase-java.lang.String-)([String](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) str) 忽略大小写后再比较大小. |
| [String](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) | [concat](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#concat-java.lang.String-)([String](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) str) 在当前字符串 后面连接str字符串,返回连接之后 的新串. |
| **boolean** | [**contains**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#contains-java.lang.CharSequence-)**(**[**CharSequence**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/CharSequence.html)**s) 判断当前字符串中是否包含s子串** |
| boolean | [endsWith](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#endsWith-java.lang.String-)([String](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) suffix) 判断当前字符串是否以suffxi结尾 |
| **boolean** | [**equals**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#equals-java.lang.Object-)**(**[**Object**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/Object.html)**anObject) 比较两个字符串是否一样** |
| boolean | [equalsIgnoreCase](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#equalsIgnoreCase-java.lang.String-)([String](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) anotherString) 忽略大小写后再判断是否一样 |
| byte[] | [getBytes](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#getBytes--)() 返回字符串在默认编码下对应的字节数组 |
| byte[] | [getBytes](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#getBytes-java.lang.String-)([String](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) charsetName) 返回字符串在指定的charsetName编码下对应的字节数组 |
| void | [getChars](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#getChars-int-int-char:A-int-)(int srcBegin, int srcEnd, char[] dst, int dstBegin) 把当前字符串[ srcBegin, srcEnd ) 范围内的字符复制到dst数组中从dstBegin开始的位置 |
| int | [indexOf](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#indexOf-int-)(int ch) 返回字符ch在当前字符串中第一次出现的索引值 |
| int | [indexOf](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#indexOf-int-int-)(int ch, int fromIndex). |
| **int** | [**indexOf**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#indexOf-java.lang.String-)**(**[**String**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html)**str) 返回字符串str在当前字符串中第一次出现的索引值** |
| int | [indexOf](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#indexOf-java.lang.String-int-)([String](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) str, int fromIndex) |
| **boolean** | [**isEmpty**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#isEmpty--)**() 判断字符串是否为空串** |
| int | [lastIndexOf](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#lastIndexOf-int-)(int ch) 返回字符ch在当前字符串中最后一次出现的索引值 |
| int | [lastIndexOf](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#lastIndexOf-int-int-)(int ch, int fromIndex) |
| int | [lastIndexOf](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#lastIndexOf-java.lang.String-)([String](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) str) 返回字符串str在当前字符串中最后一次出现的索引值 |
| int | [lastIndexOf](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#lastIndexOf-java.lang.String-int-)([String](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) str, int fromIndex) |
| **int** | [**length**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#length--)**() 返回字符串中字符的个数.** |
| boolean | [matches](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#matches-java.lang.String-)([String](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) regex) 判断当前字符串是否匹配指定的regex正则表达式 |
| [**String**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) | [**replaceAll**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#replaceAll-java.lang.String-java.lang.String-)**(**[**String**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html)**regex,** [**String**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html)**replacement) 把当前字符串中符合regex正则表达式的字符串使用replacement替换,返回替换后的新串** |
| [String](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) | [replaceFirst](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#replaceFirst-java.lang.String-java.lang.String-)([String](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) regex, [String](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) replacement) |
| [**String**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html)**[]** | [**split**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#split-java.lang.String-)**(**[**String**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html)**regex) 使用regex正则表达式分割当前字符串.返回分割后子串组成的数组** |
| boolean | [startsWith](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#startsWith-java.lang.String-)([String](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) prefix) 判断当前字符串是否以prefix开始. |
| [**String**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) | [**substring**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#substring-int-)**(int beginIndex) 返回当前字符串从beginIndex开始的子串.** |
| [**String**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) | [**substring**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#substring-int-int-)**(int beginIndex, int endIndex) 返回当前字符串[beginIndex, endIndex ) 范围内的子串** |
| char[] | [toCharArray](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#toCharArray--)() 把字符串转换为字符数组 |
| [String](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) | [toLowerCase](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#toLowerCase--)() 把字符串中的大写字母转换为小写字母 |
| [**String**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) | [**toString**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#toString--)**()** |
| [String](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) | [toUpperCase](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#toUpperCase--)()把字符串中的小写字母转换为大写字母 |
| [**String**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) | [**trim**](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#trim--)**() 去掉前后的空白字符** |
| static [String](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html) | [valueOf](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/String.html#valueOf-boolean-)(boolean b) 把其他类型的数据转换为字符串 |

#### 字符串常量池

在Java中所有使用双引号引起来的字符串字面量存储在字符串常量池中,采用享元模式.

[享元模式](https://baike.baidu.com/item/%E4%BA%AB%E5%85%83%E6%A8%A1%E5%BC%8F/10541959?fr=aladdin) 是对象池的一种实现。类似于线程池，线程池可以避免不停的创建和销毁多个对象，消耗性能。[享元模式](https://baike.baidu.com/item/%E4%BA%AB%E5%85%83%E6%A8%A1%E5%BC%8F/10541959?fr=aladdin) 也是为了减少内存的使用，避免出现大量重复的创建销毁对象的场景。（关于设计模式后续课程介绍）

#### String对象是不可变的

String字符串是不可变的,一旦创建String对象后,包含在这个对象中的字符序列是不可改变的,直到这个对象被销毁.String的concat(), toLowerCase(), trim()等方法都是返回一个新的字符串对象, 原来的字符串对象不变.使用+进行字符串连接时,总是会生成一个新的字符串对象.

### StringBuilder/StringBuffer

String字符串对象是不可变的,每次进行字符串连接都会生成新的字符串对象. 如果频繁进行字符串连接,使用StringBuilder/StringBuffer.StringBuffer/StringBuilder字符串对象是可变的,当创建StringBuilder对象后,通过append(),reverse()等方法可以改变这个字符串对象中的字符序列. 最后可以调用StringBuilder对象的toString()方法转换为String对象.区别 StringBuffer是线程安全的 ,StringBuilder不是线程安全的.如果开发的程序不是多线程使用StringBuilder

### 练习

1 从键盘上输入身份证号, 判断出生日期,性别

2 定义方法把字符串逆序

3 有字符串:

”101,lisi,98;202,wangwu,76;303,chenqi,84;404,zhangsan,49;505,xiaoming,67”,保存的学生的学号,姓名,成绩信息. 要求把字符串中学生信息取出来,创建Student对象, 把Student对象保存到数组中;遍历学生对象数组;在数组中查找名字为xiaoxiao同学是否存在.对数组中学生对象根据成绩降序排序

## 日期相关类

### Date

java.util.Date类中提供了与系统无关的用于处理日期与时间的封装,可以调用无参构造方法获得当前时间.

### LocalDateTime

Date日期不是线程安全的, java8新增了java.time包,定义一组线程安全的日期类.在多线程环境中使用LocalDateTime类获得当前日期. LocalDateTime类的构造方法使用private修饰为私有的, 该类中有一个静态方法 now(), 该方法返回当前日期对象

## 数学相关类

### Math

java.lang.Math类,定义了一组与数学函数相关的操作,包括三角函数,对数操作,指数操作等.Math类构造方法是private私有的,不能创建Math对象, Math类的所有方法都是方法,可以通过类名直接调用.

### Random

java.util.Random类,专门用于生成随机数的. 提供两个构造方法,无参构造方法使用默认的种子(当前时间), 另一个构造方法可以指定随机数的种子,相同种子的Random对象可以生成相同的随机数序列.

### DecimalFormat

java.text.DecimalFormat类可以对数字格式化, 常用的数字格式符有#与0,区别在于使用0格式符时,不足的位数会补0.

### BigDecimal

如果进行科学计算,财务计算时,使用double可能不准确, 可以使用BigInteger,BigDecimal类. 可以调用add(),subtract(),multiply(),divide()进行加减乘除操作.

## 包装类

### 包装类概述

Java为了使编程更加方便,Java为每个基本类型都提供了对应的包装类,如:

|  |  |
| --- | --- |
| 基本类型 | 包装类 |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| char | Character |
| boolean | Boolean |

包装类的继承关系如下:



### 如何创建包装类对象

所有的包装类, 都可以根据基本类型创建包装类对象.

例如：

Integer i1 = new Integer(456);

Double d1 = new Double(3.14);

### 包装类的常用方法

（1）compare()方法

（2）equals()方法

（3）parseInt()等方法

（4）valueOf()方法

### 装箱与拆箱

装箱是指把基本类型转换为包装类对象;

拆箱是指把包装类对象转换为基本类型;

Java可以自动进行装箱与拆箱

# 集合

## 集合概述

集合是用来存储引用类型数据的容器.

注意:集合只能存储引用类型数据, 不能存储基本类型数据.数组是用来存储同种类型数据的容器,数组元素既可以是基本类型,也可以是引用类型数据,而集合只能保存对象; 定义了数组后存储元素的个数就确定了,集合中存储元素的数量是可以变化的.

集合分为两大类:

（1）Collection集合,其特点是单个存储元素;

（2）Map集合, 特点是按<键,值>对的形式存储,

如存储 <学生姓名,成绩>.

## Collection集合

### Collection集合的结构

Collection集合用于存储单个的数据,主要有两个接口List与Set. List集合存储有序可重复的数据,Set集合存储无序不可重复的数据.Collection集合继续结构如图所示:



### Collection的基本操作

集合的基本操作包括:添加元素,删除元素,判断是否包含指定的元素,遍历集合中的元素.

|  |  |
| --- | --- |
| boolean | [add](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/util/Collection.html#add-E-)([E](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/util/Collection.html) e) 向集合中添加元素e |
| boolean | [addAll](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/util/Collection.html#addAll-java.util.Collection-)([Collection](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/util/Collection.html)<? extends [E](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/util/Collection.html)> c) 把参数c集合中所有的元素都添加到当前集合中 |
| void | [clear](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/util/Collection.html#clear--)() 清空集合中所有的元素 |
| boolean | [contains](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/util/Collection.html#contains-java.lang.Object-)([Object](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/Object.html) o) 判断当前集合中是否包含o |
| boolean | [isEmpty](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/util/Collection.html#isEmpty--)() 判断当前集合是否为空. |
| [Iterator](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/util/Iterator.html)<[E](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/util/Collection.html)> | [iterator](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/util/Collection.html#iterator--)() 返回集合的迭代器 |
| boolean | [remove](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/util/Collection.html#remove-java.lang.Object-)([Object](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/Object.html) o) 删除集合中第一个与o匹配的元素 |
| int | [size](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/util/Collection.html#size--)() 返回集合中元素的数量 |
| [Object](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/lang/Object.html)[] | [toArray](mk:@MSITStore:D:\course\03-JavaSE\resources\JDK8.0API.chm::/java/util/Collection.html#toArray--)() 把集合转换为数组 |

## List集合

### List基本操作

List接口继承了Collection接口,Collection有的操作List都有. List集合为每个元素指定了索引值,增加了针对索引值的操作; 在JDK8中, List增加了sort(Comparator)排序功能.

### ArrayList与Vector

ArrayList与Vector底层数据结构都是数组;

数组的初始化容量为 10;

扩容: ArrayList按1.5倍大小扩容, Vector按2倍大小扩容;

区别在于Vector是线程安全的,ArrayList不是 线程安全的

Vector与ArrayList一样，也是通过数组实现的，不同的是它支持线程的同步，即某一时刻只有一个线程能够写Vector，避免多线程同时写而引起的不一致性，但实现同步需要很高的花费，因此，访问它比访问ArrayList慢。

数组特点:访问快,添加删除慢. 数组可以通过下标快速计算出每个元素的地址(偏移量); 在添加/删除元素时可能需要扩容,复制/移动大量的元素. 如果集合就是以访问为主, 很少添加/删除,就选择ArrayList集合

### LinkedList

LinkedList底层数据结构是双向链表. 特点是访问慢,添加删除效率高. 链表每次访问时总是从头结点开始逐个访问; 在添加/删除时,只需要修改对应的指针域即可, 不需要扩容,也不需要复制/移动元素,效率高.

## Set集合

### HashSet集合的基本操作

Set集合存储特点: 无序,不可重复.

无序是指存储顺序与添加顺序可能不一样;

不可重复是指不允许存储重复的数据.

### HashSet

HashSet底层是HashMap; 向HashSet中添加元素就是把元素作为键添加到底层的HashMap中;HashSet就是HashMap键的集合. 查看HashSet源码发现在HashSet的无参构造方法中创建了HashMap集合, 所以HashSet底层是HashMap（关于底层是HashMap暂时先记住就可以了）。

### TreeSet

TreeSet实现了SortedSet接口, 可以对集合中的元素自然排序, 要求集合中的元素必须可比较大小:

(1)要么在构造方法中指定Comparator比较器

(2)如果没有在构造方法中指定Comparator, 要求元素的类实现Comparable接口

TreeSet底层是TreeMap; 向TreeSet添加元素就是把元素作为键添加到底层的TreeMap中;TreeSet就是TreeMap键的集合

## Map

### HashMap的基本操作

Map集合是按<键,值>对的形式存储数据.

HashMap的底层是哈希表，这是实例化Map集合最基础也是最常用的操作。

### Properties

HashTable类实现了Map接口，Properties继承了HashTable, 键与值都 是String字符串. 经常用来设置/读取系统属性. 经常使用的两个方法:setProperty(属性名,属性值)设置属性值,getProperty(属性名)返回属性值.

### TreeMap

TreeMap实现了SortedMap接口, 可以根据键自然排序. 要求键必须是可比较的.

1)在创建TreeMap时,通过构造方法指定Comparator比较器

2)如果没有在构造方法指定Comparator,要求键实现Comparable接口

## Collections工具类

java.util.Collections工具类定义了一组对集合的操作

## 泛型

泛型就是把数据类型作为参数传递,如在类实现Comparable接口时,通过泛型指定比较元素的数据类型,在创建集合时,通过泛型指定存储元素的数据类型. 泛型好处是在编译时可以进行数据类型检查.如:

List<String> list = new ArrayList<>();

向list集合中添加元素时,只能添加String字符串,如果添加其他的类型在编译时就报错误错误.

开发时也可能会遇到自定义泛型,包括方法泛型,类泛型. 方法泛型是在方法返回值类型前面定义,方法泛型只在当前方法中有效; 类泛型是在定义类时在类名后面定义,在整个类体中有效.

## Lambda表达式

Lambda表达式是JDK8中的一个重磅更新.语法:

(参数列表) -> { lambda体 }

说明:

1)参数列表中的类型可以省略

2)如果参数列表中只能一个参数, 小括弧可以省略

3)如果lambda体只能一条语句,大括弧可以省略, 语句后面的分号也要去掉,如果lambda体中这的一条语句是return语句时,return可以省略

# 异常

## 异常概述

在程序运行过程中,如果环境检测出一个不可能执行的操作,就会出现运行时错误.如之前遇到的异常:当对象为null,访问对象的实例成员会产生空指针异常NullPointerException, 使用越界的数组下标访问数组元素产生ArrayIndexOutOfBoundsException异常,把非数字字符串转换为数字会产生数字格式不正确异常NumberFormatException,对象向下转型可能会产生类型转换异常ClassCastException,日期转换异常ParseException,程序访问的文件不存在会产生FileNotFoundException异常等.

异常就是一个表示阻止执行正常进行的错误或情况.简单说异常就是程序运行过程中出现的不正常现象.Java把经常出现的一些异常现象进行了抽象,就形成了异常类.



Throwable类是所有异常的根. Error类描述的是内部系统错误,这样的错误很少发生,如果发生,除了通知用户终止终止程序外,几乎什么也不能做; Exception描述的是由程序或外部环境所引发的错误,这些错误可以被程序捕获或处理.

Exception异常分为两大类:编译时异常(checked检查异常)与运行时异常(unchecked免检异常).从类继承结构上看,如果异常类继承了RuntimeException类就是运行时异常,如果不是RuntimeException类的子类就是编译时异常.RuntimeException运行时异常描述的是程序设计错误,如错误的类型转换,访问数组元素时下标越界等,它反映出程序设计的逻辑错误,Java不允许编写代码捕获或声明免检异常,需要由程序员通过规范的代码避免这类异常.而编译时异常必须进行预处理,否则编译错误.

## 异常处理

### 运行时异常不需要预处理

运行时异常不需要预处理,通过规范的代码可以避免. 如定义方法计算两个整数相除的结果,代码如下:

public static void divide1( int m , int n){

int result = m / n ;

System.out.println( m + "/" + n + "=" + (m/n) );

}

在调用方法时,如果传递的第二参数为0,会产生异常导致程序中断. 整数相除当除数为0时,会产生java.lang.ArithmeticException算术异常, java.lang.ArithmeticException异常类继承了RuntimeException类,是一个运行时异常,完全可以通过程序员规范的代码避免这种异常的产生.解决这个问题的简单方法就是添加一个if语句来判断除数是否为0,代码可以修改为:

public static void divide( int m , int n){

if ( n == 0 ){

System.out.println("除数不能为0");

return; //结束当前方法的执行

}

int result = m / n ;

System.out.println( m + "/" + n + "=" + (m/n) );

}

### 受检异常的抛出处理

在Java中,当前执行的代码必属于某个方法,在方法的声明位置通过throws声明它可能抛出的受检异常类型,这称为声明异常,或称为异常的抛出处理, 这样方法的调用者会被告知有异常.如:

public void method() throws IOException{}

在 定义 方法时, 通过throws声明抛出的异常,该异常类没有继承RuntimeException类, 它就是受检异常. 在 调用该方法时, 必须对方法的受检异常进行预处理,否则编译语法错误. 预处理是指当程序运行后万一产生了异常应该怎么办, 即程序员对用户在使用该程序时可能产生的异常提出一种预处理方案, 注意: 并不是说在编译阶段发生了异常 , 所有的异常都是 在运行阶段才可能产生.

预处理方式有两种:

1. throws抛出处理
2. try..catch捕获处理

注意,通过throws抛出处理,可以避免编译语法错误,如果在程序运行过程中产生了异常,程序依然会中断.

### 异常的捕获处理

语法:

try{

把有需要预处理的受检异常的代码入在try代码块中

}catch(异常类型 e){

捕获异常,进行预处理, 程序运行后,万一产生了异常,程序员提供的一预处理方案

}finally{

经常把释放系统资源的代码放在finally代码块中

}

说明:

1. 如果 try 代码块中没有出现异常,则跳过catch子句
2. 如果try代码块的某条语句抛出一个异常,Java会跳过try代码块中剩余的语句,然后查找处理这个异常的catch子句,跳转到对应的catch子句执行
3. 异常对象包含了关于异常的有价值的信息,在catch子句中捕获了异常对象e, 可以调用getMessage()方法返回异常的信息,调用printStackTrace()打印异常的跟踪信息
4. finally子句不是必需的, 不管是否有异常产生总是会执行

异常捕获处理的作用:当try代码块的业务逻辑产生了异常后,程序会立即跳转到catch子句执行,程序不会中断.可以提高程序的健壮性

## 方法覆盖中的异常处理

如果父类方法没有抛出异常,重写后也不能抛出异常,如果父类方法抛出了异常,重写后,可以抛出相同的异常,或者子异常,或者不抛出异常.

## 自定义异常

很多时候,系统是否抛出异常,可能需要根据业务需求来定.如果程序中的数据或者执行与既定的业务需求不符,这就是一种异常,与业务需求不符的异常需要由程序员来决定抛出.抛出异常需要使用throw语句,通过throw可以抛出一个异常对象.在抛出异常时要选择合适的异常类,从而可以明确地描述该异常情况,如果系统没有提供对应的异常,程序员可以自定义异常类

自定义异常类都应继承Exception类,如果希望自定义运行时异常,则该异常类需要继承RuntimeException类.定义异常类时一般提供两个构造方法,一个是无参构造方法,一个是带有String字符串参数的构造方法,这个字符串就作为异常的描述信息.

## 异常在开发中的应用

未来的实际项目开发我们需要做以”分层”为概念的开发，每一层的职能不同，代码种类和结构也不同，出现的异常信息也不同。如果每一层都需要对异常进行手动的特殊处理，那么我们的开发工作会变得效率低下，所以一般情况下我们都是将异常抛到顶层，由顶层去做最后的处理。

# IO流

## IO概述

IO(输入/输出)是所有程序都必需的部分,通过输入允许程序读取外部数据;使用输出允许程序将数据保存到外部. Java通过IO流实现输入与输出. IO流是有起点和终点的有序的字节序列.IO流分类如下:

* 字节流/字符流, 字节流是以字节为单位读写数据, 字符流是以字符为单位读写数据;
* 输入流/输出流, 如果程序从外面读取数据就是输入流, 程序把数据保存到外面就是输出流. 输入与输出是以当前程序为参照的;
* 节点流与处理流, 直接从数据源读写数据就是节点流,处理流是对节点流的包装.

通过IO流可以读写程序外部设备上的数据. 今天只讨论与文件读写有关的几个流类, 在java.io包中,以Stream单词结尾的流类都是字节流, 以Reader结尾的流是字符输入流, 以Writer结尾的流是字符输出流.常用的IO流类有:





## FileInputStream / FileOutputStream类

FileInputStream流以Stream结尾是字节流, Input表示输入流, 直接从文件中读取数据是节点流;FileOutputStream流以Stream结尾是字节流, Output表示输出流, 直接向文件中写入数据是节点流.

在程序中读取文件的基本步骤:1)在当前程序与指定的文件之间建立流通道; 2)读写文件内容; 3)关闭流通道.

### FileInputStream类

文件输入流

### FileOutputStream类

文件输出流

### 使用FileInputStream与FileOutputStream实现文件的复制

## BufferedInputStream/BufferedOutputStream

BufferedInputStream是输入流,同时也是字节流,处理流

BufferedOutputStream是输出流,同时也是字节流,处理流

## DataInputStream/DataOutputStream

这两个流类在读写数据时,可以带有数据格式. 使用DataOutputStream可以把整数,小数,字符,布尔,字符串等数据保存到文件中, 使用DataInputStream流可以从文件中依次把整数,小数,字符,布尔,字符串再读出来.需要注意的是DataInputStream输入流的读取顺序必须与DataOutputStream写入的顺序一致,否则可能会出现数据混乱的情况.

## PrintStream

PrintStream是打印字节流, 最常用的方法是print()/println().

## ObjectInputStream/ObjectOutputStream

对象序列化是指把一个对象转换为01二进制序列;对象反序列化是指把一串01二进制序列转换为对象.

在Java中,使用ObjectOutputStream流类实现对象的序列化,把对象转换为01二进制保存到文件中, 可以简单的理解为把一个对象保存到文件中; 使用ObjectInputStream流类实现对象的反序列化,可以简单的理解为把文件中的对象读取出来

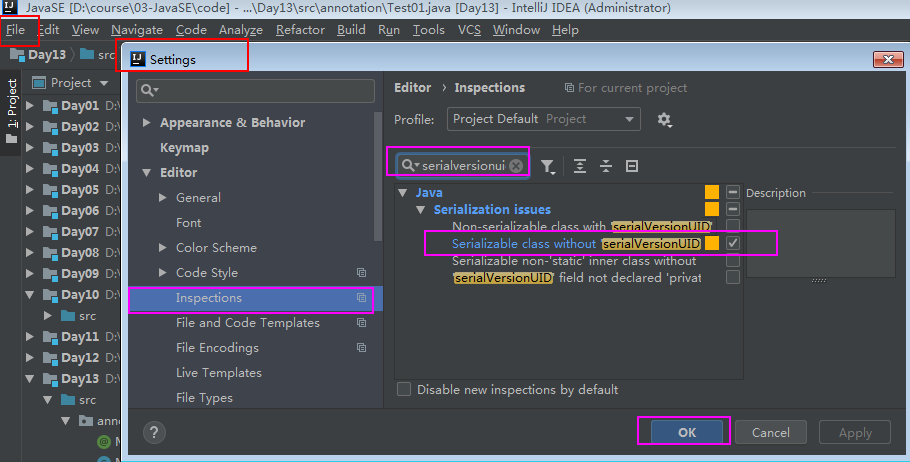
注意:

一个类的对象想要进行序列化/反序列化, 前提是该类必须实现Serializable接口. Serializable接口是一个标志接口,该接口没有任何抽象方法, 仅仅是告诉编译器,只能实现了Serializable接口的类创建的对象可以进行序列化.

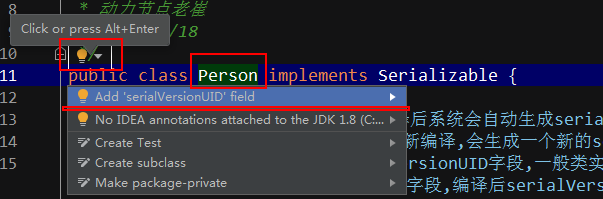
当类实现了Serializable接口后,一般会在类中添加一个序列化版本号字段:

private static final long serialVersionUID = -6846546134661332131L;

这个序列化版本号字段,可以通过IDEA工具自动生成, 打开IDEA工具的File菜单中的Settings设置对话框, 选择Editor编辑器下的Inspectons检测项, 在搜索框中输入serialversionuid, 选中Serializable class without serialversionUID后,单击OK.如图所示:



当类实现了Serializable接口后, 把光标定位到实现类时,IDEA会弹出警告,选择Add “serialVersionUID” field添加serialVersionUID序列化版本号字段, 系统就会自动生成该字段.



## FileReder/FileWriter

FileReader是字符流,是输入流,是节点流. 用来以字符为单位读取文件内容.FileWrtier是字符流,是输出流,是节点流. 用来以字符为单位把数据保存到文件中. 注意:

* 以Reader结尾的字些字符流只能读写纯文本文件
* FileReader/FileWriter读写的文本文件的编码要与当前环境的编码兼容

## InputStreamReader/OutputStreamWriter

FileReader/FileWriter只能读写与当前环境编码兼容的文件.如果读写的文件编码与当前环境编写不兼容, 使用InputStreamReader/ OutputStreamWriter. InputStreamReader可以把字节流转换为字符流,OutputStreamWriter可以把字符流转换为字节流, 这一对流类又称为转换流。

## BufferedReader/BufferedWriter

BufferedReader是字符流,输入流,处理流; BufferedWriter是字符流,输出流,处理流. 我们称之为字符缓冲流.字符缓冲流默认有8192字符大小的的缓冲区, 就是8192大小的字符数组

## 装饰者设计模式/适配器设计模式

设计模式（Design pattern）是软件开发人员在软件开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。设计模式是一套被反复使用的、多数人知晓的代码设计经验的总结。使用设计模式是为了重用代码、让代码更容易被他人理解、保证代码可靠性.

装饰者设计模式,可以对已有类的已有方法进行功能的扩展,在IO流类中以Filter开头的流类采用了装饰者设计模式.

适配器设计模式将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。在IO流类中的InputStreamReader/OutputStreamWriter转换流类就采用了适配器设计模式。

享元模式（Flyweight Pattern）主要用于减少创建对象的数量，以减少内存占用和提高性能。String字符串字面量,-128~127范围内的包装类对象就采用了享元模式

迭代器设计模式用于顺序访问集合对象的元素，不需要知道集合对象的底层表示。

策略模式就是把一系列的算法一个个封装起来, 并且使它们可相互替换

工厂模式,就是定义一个方法,方法返回值类型是接口类型,而实际返回的是接口实现类对象

关于常用设计模式，后续课程会详细介绍，此处做个了解即可。

## File类

File类也在java.io包中. 读写文件内容使用IO流, 操作文件/文件夹使用File类. 如新建,删除文件和文件夹;查看文件的属性,如查看文件夹中的内容.注意,不管是文件还是目录都是使用File操作.

### 创建File对象

File经常使用文件路径字符串来创建File对象.文件路径可以是绝对路径, 文件的绝对路径是指从根目录开始的路径 ,在Windows操作系统中就是从盘符开始的路径. File对象路径也可以是相对路径.默认情况下,相对路径是相对于用户的工作路径,即user.dir属性指定的路径

### 文件相关操作

通过File关联一个文件,可以查看文件的相关属性,包括文件大小, 最后一次修改时间,文件绝对路径等,也可以通过File创建新文件,删除文件.

### 文件夹相关操作

File关联一个文件夹,可以通过File对象创建/删除文件夹,查看文件夹的内容.

# 线程

## 线程相关概念

**进程** (Process): 是计算机中的程序关于某数据集合上的一次运行活动,是操作系统进行资源分配与调度的基本单位. 可以把进程简单的理解为正在操作系统中运行的一个程序.

**线程** (Thread): 是进程的一个执行单元. 一个线程就是进程中一个单一顺序的控制流, 是进程的一个执行分支.进程是线程的容器,一个进程至少有一个线程.一个进程中也可以有多个线程.

在操作系统中是以进程为单位分配资源,如CPU, 虚拟存储空间等. 多个线程共享同一个进程的资源.每个线程都有各自的线程栈,自己的寄存器环境,自己的线程本地存储.

**主线程:** JVM启动时会创建一个主线程,该主线程负责执行main方法 . 主线程就是运行main方法的线程.

Java中的线程不是孤立的,线程之间存在一些联系. 如果在A线程中创建了B线程, 称B线程为A线程的子线程, 相应的A线程就是B线程的父线程

**串行,并发与并行:** 串行是指各个任务依次执行,并发是指在某段时间内交替完成多个任务; 并行是同时完成多个任务.

并发可以提高事物的处理效率, 即一段时间内可以处理或者完成更多的事情.并行是一种更为严格,理想的并发. 从硬件角度来说, 如果是单核CPU,一个处理器一次只能执行一个线程的情况下,处理器可以使用时间片轮转技术 ,可以让CPU快速的在各个线程之间进行切换, 对于用户来说,感觉是三个线程在同时执行.如果是多核心CPU,可以为不同的线程分配不同的CPU内核.

## 线程的创建与启动

在Java中,创建一个线程就是创建一个Thread类(子类)的对象(实例). Thread类有两个常用 的构造方法:Thread()与Thread(Runnable).对应的创建线程的两种方式:

* 定义Thread类的子类
* 定义一个Runnable接口的实现类

这两种创建线程的方式没有本质的区别.因为Java中的类只允许单继承,当线程类继承了Thread后就不能再继承其他类了,所以开发中经常使用实现Runnable接口的形式创建线程,这样Runnable实现类还可以继承其他类,使用更灵活.

### 定义Thread子类的形式创建线程

### 实现Runnable接口的形式创建线程

### 实现Callable接口的形式创建线程

有人把实现Callable接口创建线程作为创建线程的第三种方式,其实也是调用 Thread(Runnable)构造方法创建线程对象. 与Runnable接口相比,Callable接口:

* 接口的call()方法可以有返回值,在实现Callable接口时通过泛型指定返回值类型
* call()方法声明抛出了异常

## 线程的常用方法

### currentThread()方法

Thread.currentThread()方法可以获得当前线程. Java中的任何一段代码都是执行在某个线程当中的. 执行当前代码的线程就是当前线程. 注意,同一段代码可能被不同的线程执行,因此当前线程是相对的,Thread.currentThread()方法的返回值是在代码实际运行时候的线程对象.

### setName()/getName()

thread.setName(线程名称), 设置线程名称, thread.getName()返回线程名称.通过设置线程名称,有助于程序调试,提高程序的可读性, 建议为每个线程都设置一个能够体现线程功能的名称

### isAlive()

thread.isAlive()判断当前线程是否处于活动状态,活动状态就是线程已启动并且尚未终止

### sleep()

Thread.sleep(millis); 让当前线程休眠指定的毫秒数,当前线程是指Thread.currentThread()返回的线程

### getId()

thread.getId()可以获得线程的唯一标识. 注意,某个编号的线程运行结束后,该编号可能被后续创建的线程使用,重启的JVM后,同一个线程的编号可能不一样

### yield()

Thread.yield()方法的作用是放弃当前的CPU资源,线程由RUNNING运行状态转换为READY就绪状态,重新等线程调度器调度.

### setPriority()

thread.setPriority( num ); 设置线程的优先级. java线程的优先级取值范围是 1 ~ 10 , 如果超出这个范围会抛出异常IllegalArgumentException.

在操作系统中,优先级较高的线程获得CPU的资源越多.线程优先级本质上是只是给线程调度器一个提示信息,以便于调度器决定先调度哪些线程. 注意不能保证优先级高的线程先运行.Java优先级设置不当或者滥用可能会导致某些线程永远无法得到运行,即产生了线程饥饿.线程的优先级并不是设置的越高越好,一般情况下使用普通的优先级即可,即在开发时不必设置线程的优先级.

### interrupt()

interrupt()方法可以中断线程. 注意调用interrupt()方法仅仅是在当前线程打一个中断标志,并不是真正的停止线程. 调用Thread.interrupted()静态方法可以判断当前线程的中断状态,也可以调用thread.isInterrupted()实例方法判断当前线程的中断状态. 这两个方法的区别在于: 静态方法interrupted()判断完中断状态后会清除中断标志, 实例方法isInterrupted()判断完后不会清除中断标志.

### setDaemon()

Java中的线程分为用户线程与守护线程. 守护线程是为其他线程提供服务的线程,如垃圾回收器(GC)就是一个典型的守护线程. 守护线程不能单独运行, 当JVM中没有其他用户线程,只有守护线程时,守护线程会自动销毁, JVM会退出. 调用setDaemon(true)可以把线程设置为守护线程

### join()

在当前线程中加入另外一个线程,当前线程转为等待状态,当被加入的线程执行完毕后,当前线程转为RUNNABLE可运行状态.

## 线程的生命周期

线程的生命周期是线程对象的生老病死,即线程的状态.线程生命周期可以通过getState()方法获得, 线程的状态是Thread.State枚举类型定义的, 由以下几种:

NEW,新建状态. 创建了线程对象,在调用start()启动之前的状态;

RUNNABLE,可运行状态. 它是一个复合状态,包含:READY和RUNNING两个状态. READY状态该线程可以被线程调度器进行调度使它处于RUNNING状态, RUNING状态表示该线程正在执行. Thread.yield()方法可以把线程由RUNNING状态转换为READY状态

BLOCKED阻塞状态.线程发起阻塞的I/O操作,或者申请由其他线程占用的独占资源,线程会转换为BLOCKED阻塞状态. 处于阻塞状态的线程不会占用CPU资源. 当阻塞I/O操作执行完,或者线程获得了其申请的资源,线程可以转换为RUNNABLE.

WAITING等待状态. 线程执行了Object.wait(), thread.join()方法会把线程转换为WAITING等待状态, 执行object.notify()方法,或者加入的线程执行完毕,当前线程会转换为RUNNABLE状态

TIMED\_WAITING状态,与WAITING状态类似,都是等待状态.区别在于处于该状态的线程不会无限的等待,如果线程没有在指定的时间范围内完成期望的操作,该线程自动转换为RUNNABLE

TERMINATED终止状态,线程结束处于终止状态



## 多线程编程的优势与存在的风险

多线程编程具有以下优势:

* 提高系统的吞吐率(Throughout). 多线程编程可以使一个进程有多个并发(concurrent,即同时进行的)的操作
* 提高响应性(Responsiveness).Web服务器会采用一些专门的线程负责用户的请求处理,缩短了用户的等待时间
* 充分利用多核(Multicore)处理器资源. 通过多线程可以充分的利用CPU资源

多线程编程存在的问题与风险:

* 线程安全(Thread safe)问题.多线程共享数据时,如果没有采取正确的并发访问控制措施,就可能会产生数据一致性问题,如读取脏数据(过期的数据), 如丢失数据更新.
* 线程活性(thread liveness)问题.由于程序自身的缺陷或者由资源稀缺性导致线程一直处于非RUNNABLE状态,这就是线程活性问题.
* 上下文切换(Context Switch). 处理器从执行一个线程切换到执行另外一个线程.
* 可靠性. 可能会由一个线程导致JVM意外终止,其他的线程也无法执行.

## 线程安全问题

非线程安全主要是指多个线程对同一个数据进行操作时,会出现值被更改,值不同步的情况.线程安全问题表现为三个方面: 原子性,可见性和有序性.

## 线程同步

### 线程同步机制简介

线程同步机制是一套用于协调线程之间的数据访问的机制.该机制可以保障线程安全.

Java平台提供的线程同步机制包括: 锁, volatile关键字, final关键字,以及相关的API,如Object.wait()/Object.notify()等

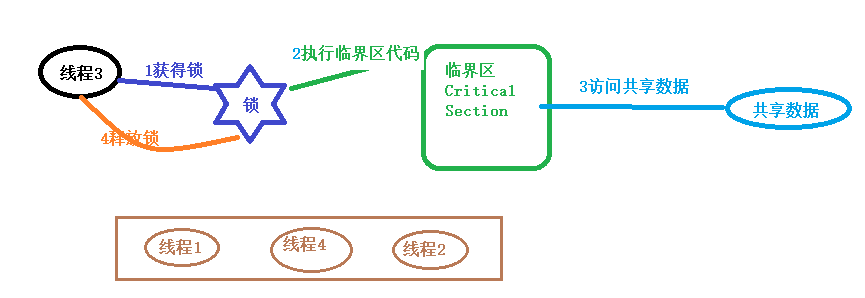
### 锁概述

线程安全问题的产生前提是多个线程并发访问共享数据.将多个线程对共享数据的并发访问转换为串行访问就不会有线程安全问题,即一个共享数据一次只能被一个线程访问.锁就是利用这种思路来保障线程安全的.

锁(Lock)可以理解为对共享数据进行保护的一个许可证. 对于同一个许可证保护的共享数据来说,任何线程想要访问这些共享数据必须先持有该许可证,一个线程只有在持有许可证的情况下才能对这些共享数据进行访问;一个许可证一次只能被一个线程持有; 许可证线程在结束对共享数据的访问后必须释放其持有的许可证.

一线程在访问共享数据前必须先获得锁; 获得锁的线程称为锁的持有线程; 一个锁一次只能被一个线程持有. 锁的持有线程在获得锁之后 和释放锁之前这段时间所执行的代码称为临界区(Critical Section).

锁具有排他性(Exclusive), 即一个锁一次只能被一个线程持有.这种锁称为排它锁或互斥锁(Mutex).



JVM把锁分为内部锁和显示锁两种. 内部锁通过synchronized关键字实现; 显示锁通过java.concurrent.locks.Lock接口的实现类实现的.

锁可以实现对共享数据的安全访问. 保障线程的原子性,可见性与有序性.

锁是通过互斥保障原子性. 一个锁只能被一个线程持有, 这就保证临界区的代码一次只能被一个线程执行.使得临界区代码所执行的操作自然而然的具有不可分割的特性,即具备了原子性.

可见性的保障是通过写线程冲刷处理器的缓存和读线程刷新处理器缓存这两个 动作实现的. 在java平台中,锁的获得隐含着刷新处理器缓存的动作, 锁的释放隐含着冲刷处理器缓存的动作.

锁能够保障有序性.写线程在临界区所执行的在读线程所执行的临界区看来像是完全按照源码顺序执行的.

注意:使用锁保障线程的安全性,必须满足以下条件:

* 这些线程在访问共享数据时必须使用同一个锁
* 即使是读取共享数据的线程也需要使用同步锁

### 内部锁:synchronized关键字

Java中的每个对象都有一个与之关联的内部锁(Intrinsic lock). 这种锁也称为监视器(Monitor), 这种内部锁是一种排他锁,可以保障原子性,可见性与有序性.

内部锁是通过synchronized关键字实现的.synchronized关键字修饰代码块,修饰该方法.

修饰代码块的语法:

synchronized( 对象锁 ) {

同步代码块,可以在同步代码块中访问共享数据

}

修饰实例方法就称为同步实例方法

修饰静态方法称称为同步静态方法

## 生产者消费者设计模式

生产者-消费者模式是一个十分经典的多线程并发协作的模式，解决多线程之间的数据平衡问题.

所谓生产者-消费者问题，实际上主要是包含了两类线程，一种是生产者线程用于生产数据，另一种是消费者线程用于消费数据，为了解耦生产者和消费者的关系，通常会采用共享的数据区域，就像是一个仓库，生产者生产数据之后直接放置在共享数据区中，并不需要关心消费者的行为；而消费者只需要从共享数据区中去获取数据，就不再需要关心生产者的行为。但是，这个共享数据区域中应该具备这样的线程间并发协作的功能：

* 如果共享数据区已满的话，阻塞生产者继续生产数据放置入内；
* 如果共享数据区为空的话，阻塞消费者继续消费数据

## Timer类

通过Timer类可以完成定时任务调度.