# 面向对象基础

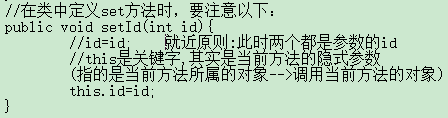
## 一、类

在定义一个类时，当类是描述具体事务的时候，该类不应该具有主方法，即定义类时要在main主方法之外。类里面有属性和行为，属性用变量代替，称为**成员变量**，成员变量在声明时若不赋值，其值为对应数据类型的默认值。行为用方法代替，称为**成员方法**。定义类名时，首字母大写，定义方法时首字母小写。

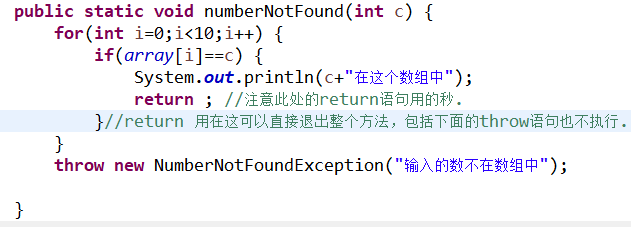
Java中parameter和arguments的区别：两者都是参数的意思，但parameter一般指函数定义中的参数，即形参；而arguments一般指函数调用中的参数，即实参。

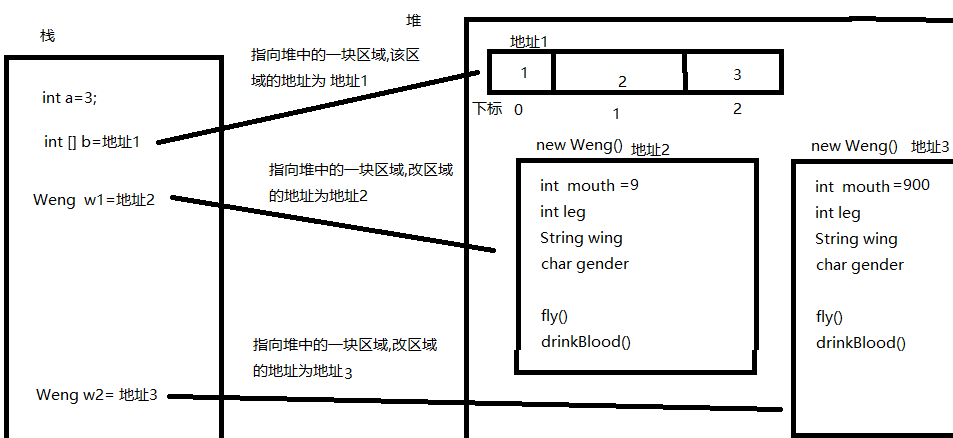
**封装**：**将属性私有化，然后提供对外的公共方法访问**。要使外部的方法调用该类的属性，需要在类的封装里面同时对每个属性声明get、set方法。其中，get方法有返回值，set方法没有返回值。

定义完类之后，在主函数中创建对象。要调用类中的私有属性，要使用get、set方法，对类的属性赋值用如：Object.setName(“xxx”)，而调用类的公开属性时对其赋值用如：Object.name=” ”。



**return 语句有两个作用**：①返回值②跳出整个语句，即在一个方法中，若在for语句中有return语句，而在for之外还有语句也不执行，而且是整个方法内的语句也不执行。





## 二、构造方法（构造器）

如：**public 方法名(){……}**。**构造器用于类的对象的属性初始化。**

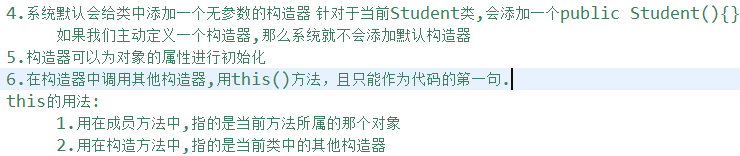
**特点：**

1.没有返回值，且连void都没有。public int student(){…}和public void student(){…}都不是构造器，public Person(){…}才是构造器。普通方法有返回值类型（void返回值为空），而构造器方法没有。

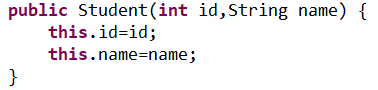
2.构造器的方法名和类名要完全一致。类名首字母一般大写，则构造器的方法名一般也是大写的。

3.构造器在创建对象时自动被调用，创建一个对象就会调用一次

4.**在其他构造器中调用构造方法，用this()，括号里面是参数，且必须放在第一位**。



有参构造方法的用法：如下



**构造器可以用private修饰，如private Person(){….}，但是该构造器不能在其他类中创建对象**，即在其他类中使用Person() p=new Person()会报错，显示该构造器的对象不可见。

**构造器的权限和该类的权限的可以是不一致的**，比如单例模式中构造器的权限是private修饰的，该类的对象不能在外部创建，只能在内部的静态方法中创建对象，然后外部调用这个类的静态方法获取对象。

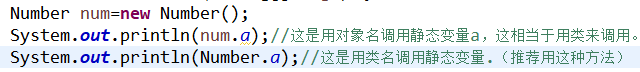
java中的普通方法名可以和类名是一样的，但是最好不要写一样的。

## 三、变量

### 1.静态变量

用static修饰的数据为静态数据。**特点：**

1.静态数据用**类名.数据名(方法、变量)**调用，通过对象名调用其实也是在用类名调用。如类中的静态变量：，在主函数main()中调用时：



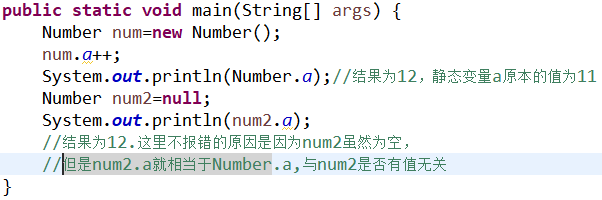
2.静态变量不能在main()方法和其它方法中定义，能在类中定义。

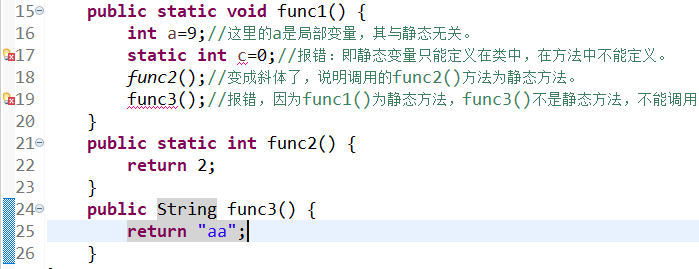
包的作用是解决命名冲突问题。

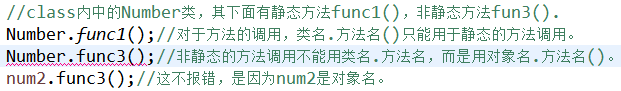
3.静态变量是属于类的，与对象无关，在每个对象之间是**共享**的。

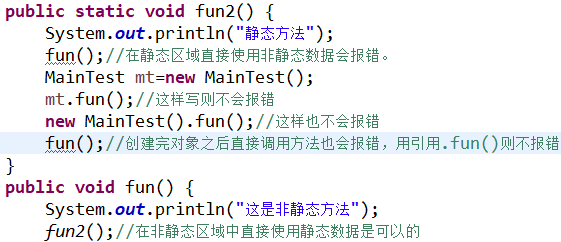
4.**在静态区域中不能直接使用非静态数据，但非静态区域中可以直接使用静态数据**。**在初始化中是先静态后非静态**，所以在静态区域中是不能直接使用非静态的数据，因为非静态的数据还没有。但是非静态方法如果是用new创建对象之后，在静态方法中用对象引用调用非静态方法则是可以的，但是若创建完对象之后直接在静态区域调用方法(不用对象引用调用)则还是会报错。

**不管在什么方法中都不能用static修饰变量，因为static修饰的数据是属于类的，应该在类中修饰成员变量，而不能用static修饰方法的局部变量**。

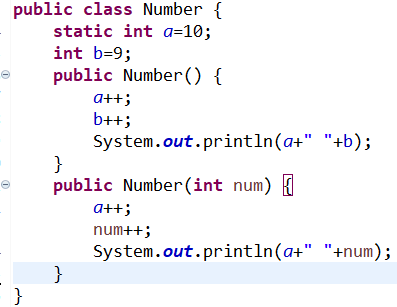


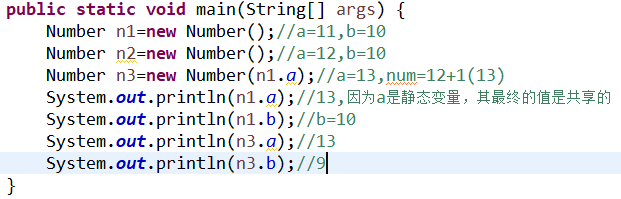




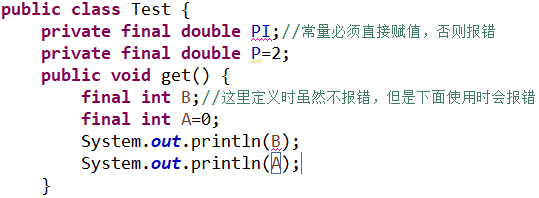


5.(**特别注意**)**静态区域中不能使用this关键字**：因为this关键字指的是当前方法所属的那个对象，而静态方法不属于任何对象，其属于类。





final修饰变量时，为常量。**常量只能赋值一次，且不能修改**。**定义常量的时候必须直接赋值，否则报错**。可用final修饰static、也可修饰局部变量。



### 2.变量总结

成员变量：定义在类中的属性且不被static修饰的变量，作用域为整个类。

静态变量：用static修饰的变量，作用域在类中的静态区域。

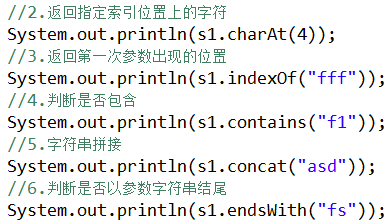
局部变量：不能用权限修饰(如private、public、protected)，不能用static修饰。在函数中定义的变量，作用域只在函数中，定义在函数中的参数上的变量，定义在for循环内部的变量。

## 四、字符串([4.字符串的创建方式四、字符串](#_4.字符串的创建方式))

### 1.定义字符串

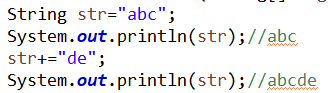


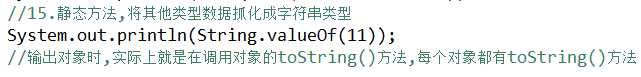
### 2.字符串的方法

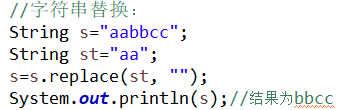


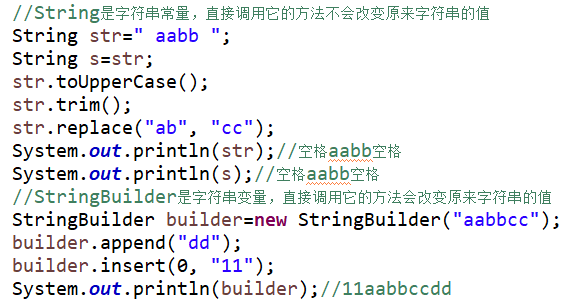


注意上面的**substring(beginIndex,endIndex)**：是包含beginIndex所有所在的字符，但不包含endIndex索引所在的字符，即左闭右开的。substring(beginIndex)：从起始索引位置开始一直到字符串的尾部。





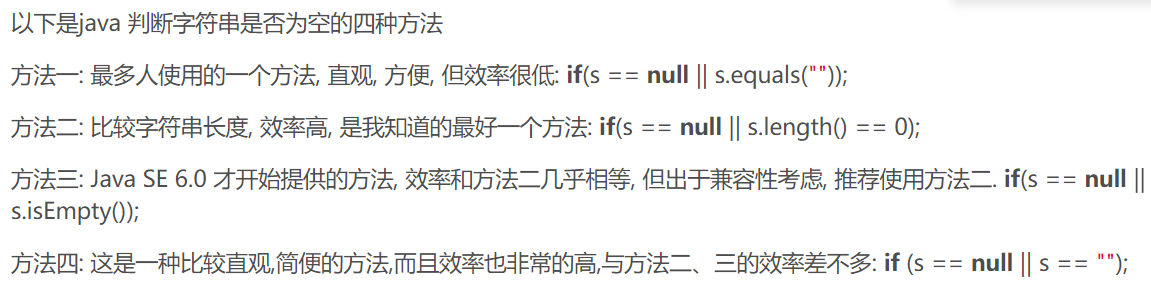




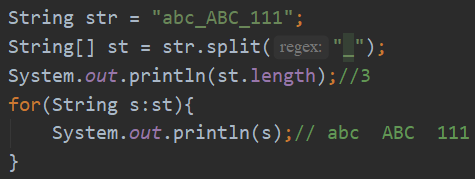
String**.replace(CharSequence target，CharSequence replace)**：返回值还是String类型。第一个参数为要被替换掉的字符串，第二个参数为替换的字符串。

当s1**.indexOf(“xxx”)**的返回值为-1时表示没有这个字符串。

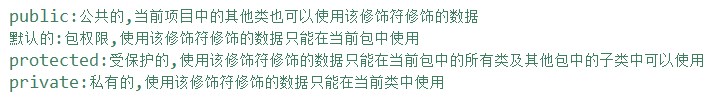
注意求数组的长度时，用ArrayObject.length；求字符串的长度时，用StringObject.length()，字符串的长度带了括号。

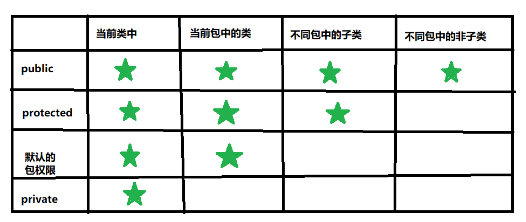


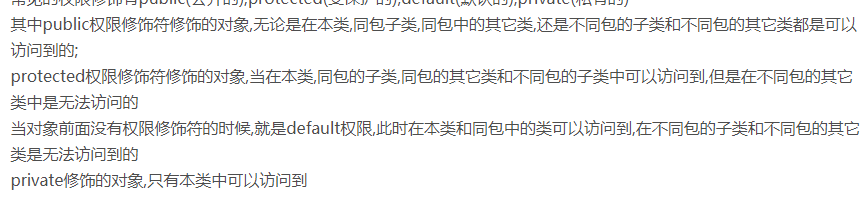
**split()**方法：分割字符串，结果返回一个字符串数组。



## 五、权限







**注意外部类只能用public 、abstract 、final和默认修饰。不能用protected、private、static修饰外部类，但是可以修饰内部类。**

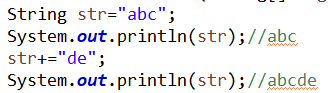
## 六、String、StringBuilder、StringBuffer的区别

主要区别在于两方面：运行速度和线程安全。

### 1.运行速度

**运行速度(执行速度)由快到慢：StringBuilder>StringBuffer> String**。String为**字符串常量**，其一旦创建就不可修改。StringBuilder和StringBuffer为**字符串变量**，可修改。

下面这段String代码看似修改了实际没有，其实际运行机制为：JVM先创建一个str对象，然后把”abc”赋值给str。然后第三行代码，JVM又新建了一个str对象，把原来旧的str的值和”de”值加起来一起赋值给新的str对象。而旧的str被JVM的垃圾回收机制给回收了。故java中的String对象的实际操作就是一个不断创建新的对象并且把旧的对象回收的过程，所以执行速度慢。



而StringBuilder和StringBuffer的对象则是变量，对变量进行操作就是直接对该对象进行修改，而不进行重新创建和回收，故速度比String要快。注意若将上面的String代码该成下面这种，则其速度要比StringBuilder和StringBuffer要快，下面这种的效果和String str=”abcde”的效果一样。



### 2.线程安全

如果一个StringBuffer对象在字符串缓冲区被多个线程使用时，StringBuffer中很多方法可以带有synchronized关键字，所以可以保证线程是安全的，但StringBuilder的方法则没有该关键字，所以不能保证线程安全，有可能会出现一些错误的操作。若进行多线程的操作，则使用StringBuffer，若是单线程，则使用速度更快的StringBuilder。

### 3.小结

String 适合在少量的字符串操作的情况。

StringBuilder：适合在单线程下在字符缓冲区中进行大量的操作，比较节省资源

StringBuffer：适合在多线程下在字符缓冲区中进行大量的操作，比较耗费资源。

StringBuilder和StringBuffer是放在堆里面，而String直接赋值的是放在字符串常量池中的，new 出来的则是放在堆中的。

有空看一下String、StringBuffer、StringBuilder中的源码。

## 七、==和equals()的区别

### 1.==

在基本数据类型中，==比较的是两个的值是否相等。

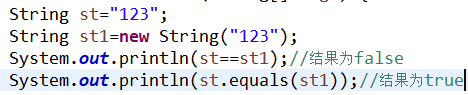
在引用数据类型中，其作用是判断两个对象的内存地址是否相等，即判断两个对象是不是指向的都是同一个对象，**比较的是内存地址**。

### 2.equals()

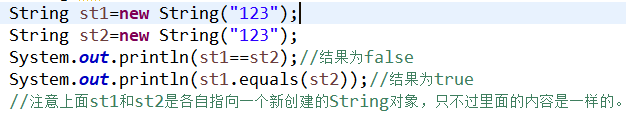
不能用于基本数据类型进行判断是否相等。

若没有对equals()方法进行重写，则比较的是引用类型变量所指向的内存地址。

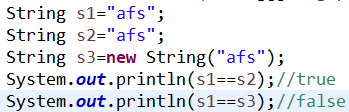
若对equals()方法进行了重写，如**String类**、Date类、包装类，则比较的是所指向的对象的内容。







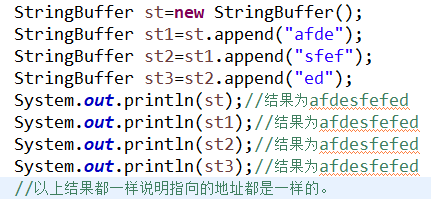
注意上面的st所指向的字符串是放在字符串常量中，而st1所指向的字符串是放在堆中。用new新建的对象都是放在堆里面。



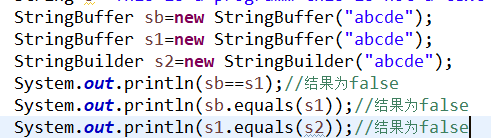
**当字符串是通过直接赋值的形式得到，则是存储在字符串常量池中，可以直接用==比较，因为它们指向的都是同一个对象。当字符串是通过new的方法得到，则不能用==比较，因为它们是存储在堆中，指向的是不同的对象**。

### 3.小结

**存在字符串常量池中的字符串可以用==比较，new出来的字符串只能用equals()方法比较。(隐性规定)所有字符串的比较都要用equals()方法**。



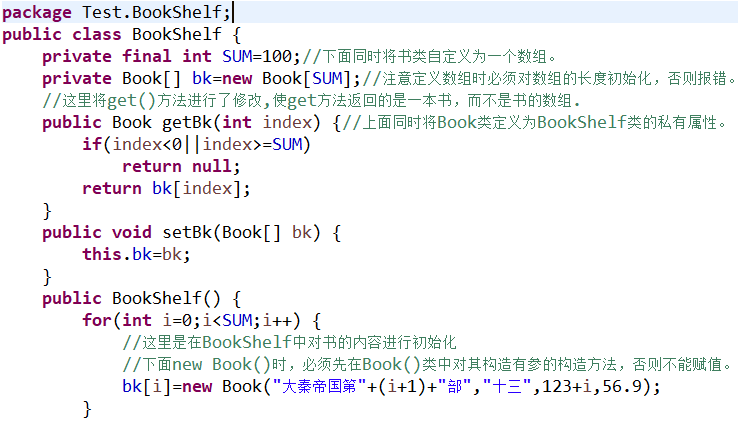
将上面的StringBuffer替换成StringBuilder的效果是一样的。



**StringBuilder和StringBuffer不能用==和equals来比较字符串的内容**。因为引用数据类型中，只有String的equals方法进行了重写，其equals()不是用==比较的，而是一个一个判断字符的内容是否相等，其他的如StringBuilder的equals()方法都是用==比较，相当于==，在引用数据类型中==是用来比较内容的地址的。**对于比较StringBuilder、StringBuffer中的字符串内容是否相等，可以调用它们的toString()方法转换成String类型，然后用equals()方法进行比较**。

## 八、对象数组

例题：书架上有100本书，书有书名、作者、isbn、价格，从书架上找到一本书，并知道书的具体内容如书名、作者等。



## 九、垃圾清理机制

**1.判断对象是否为垃圾对象有两种方式**：

**①引用计数器法**：当一个对象被引用，则该对象的引用计数器就加1，失去一个引用就减1，然后清理掉计数器为0的对象。该方法的**弊端**：比如两个对象互相引用 ，但程序中又没有使用这两个对象，因此这两个对象不能被标记为垃圾对象。

**②GCRoot方法**：从对象出发，能找到一个GCRoot就认为该对象有用，否则就是垃圾。

**2.清理垃圾的方法**：

**①标记清理法**：只要发现垃圾内存时，标记为可回收的状态，待会一起清理掉。**弊端**：清理的内存空间多是零碎的，会产生很多空间碎片。

**②标记复制法**：将可用内存按容量划分为大小相等的两块，每次只使用其中一块。当这块内存需要进行垃圾回收时，会将此区域还存活着的对象复制到另一块上面，然后再把已经使用过的内存区域一次清理掉。浪费了一定的空间，但是速度快。

**③标记整理法**：标记整理的过程与标记清理法的过程基本类似，不过标记整理不是直接将垃圾对象进行清理，而是将所有存活对象移动到一端，然后清理掉其他内存。速度相对比较慢。

**标记清除算法仅对不存活的对象进行处理，剩余存活对象不做任何处理，造成内存碎片；而标记整理算法不仅对不存活对象进行处理清除，还对剩余的存活对象进行整理，重新整理，因此其不会产生内存碎片**。

**finalize是Object的方法，当垃圾回收器将要回收对象所占内存之前会调用这个方法，这个方法只会在对象被回收前调用一次**。

c的垃圾回收是人工的，工作量大，但是可控性高。

java中的垃圾回收是自动化的，但是可控性很差，甚至有时会出现内存溢出的情况，内存溢出也就是jvm分配的内存中对象过多，超出了最大可分配内存的大小。

GC(Garbage Collection)：垃圾回收器。垃圾回收机制的好处：**有效防止内存泄漏**。垃圾回收器通常作为一个单独的低级别的线程运行。

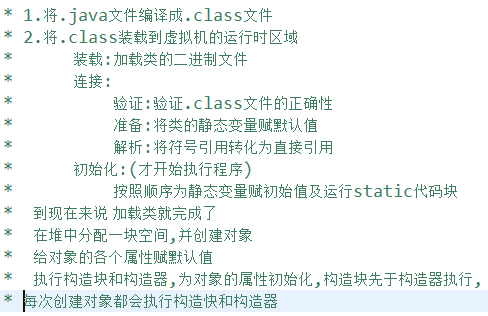
提到java的垃圾回收机制就不得不提一个方法：**手动执行System.gc()方法可以通知虚拟机进行垃圾回收，但并不能保证垃圾回收器马上进行垃圾回收**。

　System.gc()用于调用垃圾收集器，在调用时，垃圾收集器将运行以回收未使用的内存空间。它将尝试释放被丢弃对象占用的内存。然而System.gc（）调用附带一个免责声明，无法保证对垃圾收集器的调用。所以System.gc()并不能说是完美主动进行了垃圾回收。

## 十、类的加载过程和对象的初始化过程

### 1.简要过程

**创建对象时会加载类，且仅加载一次，使用静态数据也会加载类**。如Student st=new Student().**若定义了该类的数组类型，则不会加载类,如Student[] str=new Student[]**。



**初始化顺序六字口诀：先静态后非静态，先属性后方法，先声明后赋值，先父类后子类。**

**静态代码块只执行一次，因为静态代码块属于类，而加载类只加载一次**。

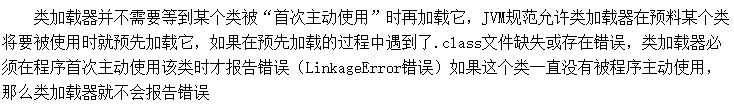
**注意：在加载类时，不管是创建对象还是使用静态数据，都只加载类一次。而在创建对象的时候还会执行构造块和构造器，只有创建对象才会执行构造块和构造器，而使用静态数据则不会执行构造快和构造器（如Student.a）。**

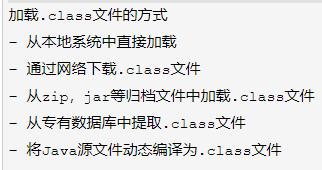
**加载类的方法：①反射中两种方法：Class.forName()和对象引用.getClass()获取Class对象；②创建对象；③执行静态代码块；④执行主函数main()；⑤创建子类的对象，会加载子类同时也会加载子类的父类**。

### 2.详细过程

**类的加载**：指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区中创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。

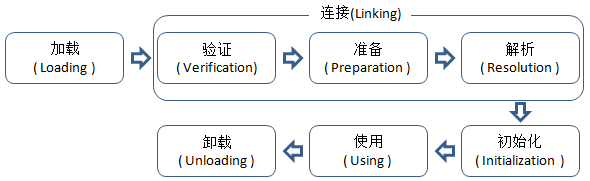
类的加载的最终产物是位于堆区中的Class对象，Class对象封装了类在方法区内的数据结构，并且向java程序员提供了访问方法区内的数据结构的接口。





类加载的过程包括：**加载、验证、准备、解析、初始化**这五个过程。

在这五个阶段中，加载、验证、准备和初始化这四个阶段发生的顺序是确定的，而解析阶段则不一定，它在某些情况下可以在初始化阶段之后开始，这是为了支持Java语言的运行时绑定（也称为动态绑定）。另外注意这里的几个阶段是按顺序开始，而不是按顺序进行或完成，因为这些阶段通常都是互相交叉地混合进行的，通常在一个阶段执行的过程中调用或激活另一个阶段。



**1.加载：查找并加载类的二进制数据**

类加载的第一个阶段就是加载，在加载阶段虚拟机需要完成下面三件事：

1.通过一个类的全限定名来获取其定义的二进制字节流。

2.将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。

3.在Java堆中生成一个代表这个类的java.lang.Class对象，作为对方法区中这些数据的访问入口。

以使用系统提供的类加载器来完成加载，也可以自定义自己的类加载器来完成加载

加载阶段完成后，虚拟机外部的二进制字节流就按照虚拟机所需的格式存储在方法区之中，而且在Java堆中也创建一个java.lang.Class类的对象，这样便可以通过该对象访问方法区中的这些数据。

**2.连接**：

(1)**验证：确保被加载的类的正确性**

(2)**准备：为类的静态变量分配内存，并将其初始化为默认值**

(3)**解析：把类中的符号引用转换为直接引用**

(1)**验证过程**目的是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。验证阶段非常重要，但不是必须的，它对程序运行其没有影响。

验证阶段大致会完成4个阶段的检验动作：

**文件格式验证**：验证字节流是否符合Class文件格式的规范；

**元数据验证**：对字节码描述的信息进行语义分析（注意：对比javac编译阶段的语义分析），以保证其描述的信息符合Java语言规范的要求

**字节码验证**：通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。

**符号引用验证**：确保解析动作能正确执行。

(2)**准备阶段**是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中分配。

对于该阶段有以下几点需要注意：

①这时候进行内存分配的仅包括类变量（static），而不包括实例变量，实例变量会在对象实例化时随着对象一块分配在Java堆中。

    ②这里所设置的初始值通常情况下是数据类型默认的值，而不是在Java代码中被显式地赋予的值。

   假设一个类变量的定义为：public static int value = 3，那么变量value在准备阶段过后的初始值为0，而不是3，因为这时候尚未开始执行任何Java方法，而把value赋值为3的putstatic指令是在程序编译后，存放于类构造器<clinit>()方法之中的，所以把value赋值为3的动作将在初始化阶段才会执行。

③**如果类字段的字段属性表中存在ConstantValue属性，即同时被final和static修饰，那么在准备阶段变量value就会被初始化为ConstValue属性所指定的值**。

   假设上面的类变量value被定义为： public static final int value = 3，编译时Javac将会为value生成ConstantValue属性，在准备阶段虚拟机就会根据ConstantValue的设置将value赋值为3。可以理解为static final常量在编译期就将其结果放入了调用它的类的常量池中。

对于同时被static和final修饰的常量，必须在声明的时候就为其显式地赋值，否则编译时不通过；而只被final修饰的常量则既可以在声明时显式地为其赋值，也可以在类初始化时显式地为其赋值，总之，在使用前必须为其显式地赋值，系统不会为其赋予默认零值。

(3)**解析阶段**是虚拟机将常量池内的符号引用替换为直接引用的过程，解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。

**符号引用**就是一组符号来描述目标，可以是任何字面量。

**直接引用**就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。

**3.初始化：为类的静态变量赋予正确的初始值**

在Java中对类变量进行初始值设定有两种方式：

   ①声明类变量时指定初始值

    ②使用静态代码块为类变量指定初始值

JVM初始化步骤：

  ①若这个类还没有被加载和连接，则程序先加载并连接该类

  ②若该类的直接父类还未被初始化，则先初始化其直接父类

  ③若类中有初始化语句，则系统依次执行这些初始化语句

类初始化时机：只有当对类的主动使用的时候才会导致类的初始化，**类的主动使用包括以下六种，即如下六种会使类初始化**：

①创建类的实例，即new的方式

②访问某个类或接口的静态变量，或者对该静态变量赋值

③调用类的静态方法

④反射中的两个获得对象的方法，Class.forName()和getClass

⑤初始化某个类的子类，则其父类也会被初始化

⑥执行main()方法

**结束生命周期**

在如下几种情况下，Java虚拟机将结束生命周期：

①执行了System.exit()方法

②程序正常执行结束

③程序在执行过程中遇到了异常或错误而异常终止

④由于操作系统出现错误而导致Java虚拟机进程终止

**类加载器**

可以大致划分为以下三类

①**启动类加载器**：Bootstrap ClassLoader

②**扩展类加载器**：Extension ClassLoader

③**应用程序类加载器**：Application ClassLoader

**类的加载**

类加载有三种方式：

①命令行启动应用时候由JVM初始化加载

②通过Class.forName()方法动态加载

③通过ClassLoader.loadClass()方法动态加载

**总结：**

**类的加载五个阶段：**

**1.加载：查找并加载类的二进制数据**

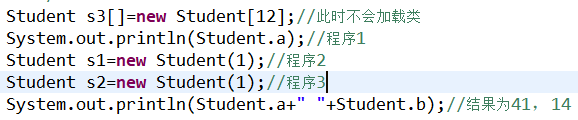
**2.连接**：

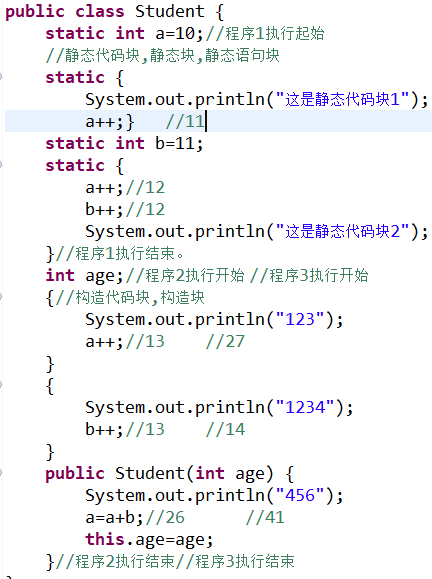
**(1)验证：确保被加载的类的正确性**

**(2)准备：为类的静态变量分配内存,并将其初始化为默认值**

**(3)解析：把类中的符号引用转换为直接引用**

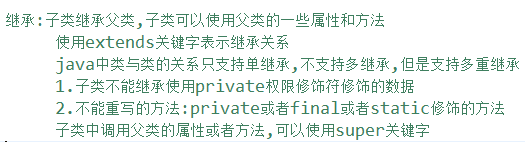
**3.初始化：为类的静态变量赋予正确的初始值**





## 十一、继承

类与类之间用**extends**表继承



### 1.方法重写

**子类继承父类，子类同父类的方法名相同，参数列表相同，返回值相同，子类方法的权限不能小于父类方法的权限，子类的异常不能大于父类的异常，且这个方法不能被static、private、final修饰**。

### 2.方法重载

**同一个类中，方法名相同，参数列表不同（类型、顺序、个数），与返回值和权限无关**。

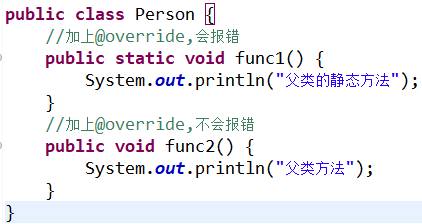
### 3.不能继承使用的数据

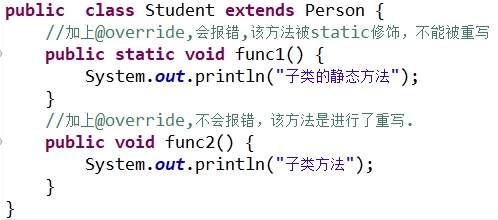
**子类可以继承使用父类静态的属性和方法，可以继承protected、public、默认状态修饰的数据，但不能继承使用private修饰的数据、子类不能继承final修饰的类**。

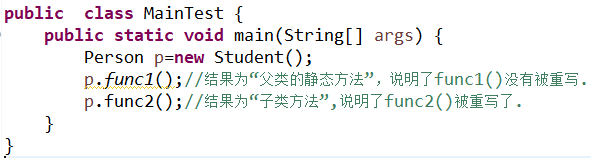
### 4.不能被重写的方法

**private**修饰的方法、被**final**修饰的方法、静态**static**方法也不能被重写。如父类的方法：private void smoke(){….},子类用public void smoke(){….}会报错。

特别注意：**静态方法不能被重写**。Java中如果父类中含有一个静态方法，且子类中也含有一个返回值类型、参数列表、方法名完全相同的静态方法，该子类实际上是对父类中的该同名方法进行了隐藏，而非重写。也就是说父类中的这个静态方法和子类中的这个静态方法实际上是没有任何关系的，它们的行为也不具有多态性。







### 5.final的用法

**final修饰常量不能被再次赋值，修饰类则不能被继承，修饰方法则不能被重写**。

### 6.super和this

**子类调用父类的属性和方法用super，注意不能调用父类中被private修饰的属性和方法**，如调用父类的成员变量a和成员方法smoke()，在子类中用super.a和super.smoke()。

**this()调用本类中的构造器，super()调用父类的构造器。this()和super()都必须放在第一位，因此这两者不能一起用。**

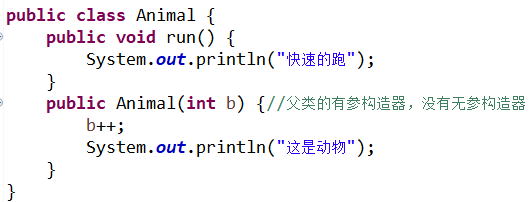
### 7.小结

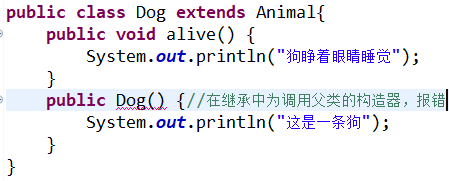
**注意外部类只能用public 、abstract 、final和默认修饰。不能用protected、private、static修饰外部类，但是可以修饰内部类。**

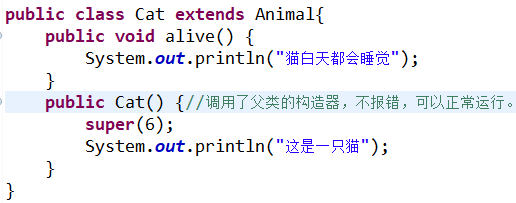
**不能继承**：private修饰的数据(属性、方法)、final修饰的类(但可以继承final修饰的属性、方法)、 (也可以继承static修饰的属性、方法)。**注意父类的构造方法是不能被子类继承的**。

**不能被重写的方法**：private修饰的方法、static修饰的方法、final修饰的方法。

**在继承中，子类的构造器中，必须调用一个父类的构造器**，若不调用的话，系统会报错。一个类的构造器可以有多个。默认情况下是系统自动用super(无参)调用父类的无参构造器，但若父类中没有无参构造器，则要自己调用。

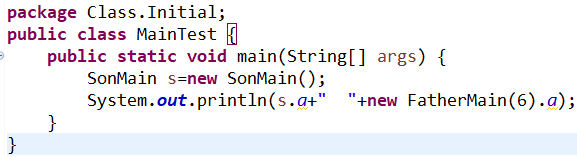


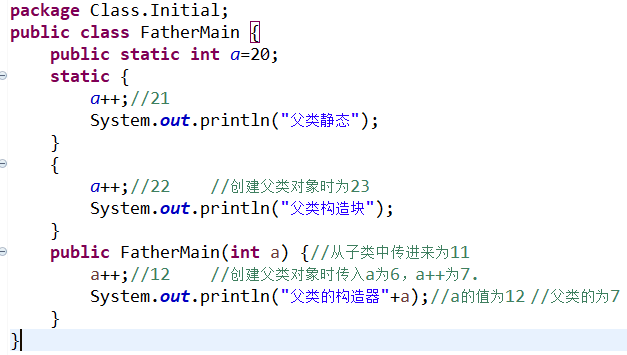


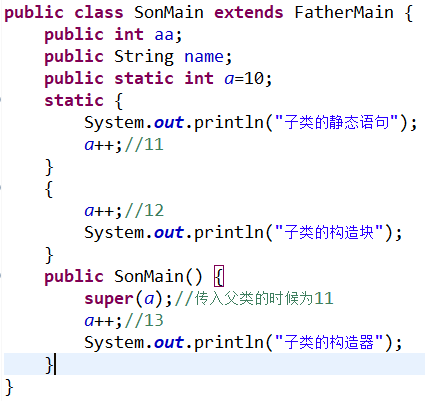


## 十二、继承的对象初始化

先加载父类，执行父类的静态代码块，然后加载子类，执行子类的静态代码块。若创建了子类的对象，则再继续加载父类的构造块和构造器，再加载子类的构造块和构造器（实际上是**先执行子类的构造块和构造器中的super()函数**，执行完之后子类构造器中的super()函数没有了，然后再跳转到父类的构造块和构造器，执行完之后，再执行子类中的构造块和构造器的内容）。类中的构造块和构造器是同时执行的，其中子类构造器中的super()函数是先调用，调用完之后，再执行子类构造块和构造器中的其他内容。







## 十三、多态

**多态**：对于同一消息做出不同反应。

### 1.多态的要素

**①有继承关系，子类继承父类**

**②子类重写父类的方法**

**③父类引用指向子类对象**，如Person p=new Student(); Person p是父类引用，new Student()是子类对象。

**④父类引用调用重写的方法**。

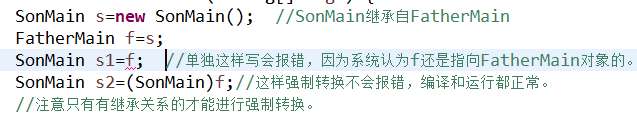
**静态方法是没有多态的，因为静态方法不能被重写**。

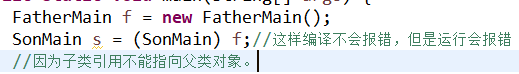
### 2.多态的实现机制：绑定

**①前期绑定(静态绑定，编译期绑定)**：所有属性、构造器、private或者static或者final修饰的方法都属于前期绑定。也就是说父类引用虽然指向了子类对象，但其指向的比如属性、构造器等都是父类的。

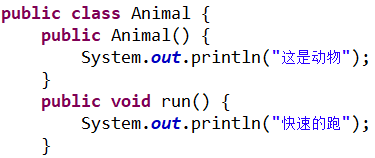
**②后期绑定(动态绑定，运行时绑定)**：在Person p=new Student();中，**只有子类重写了父类的方法，才会调用子类重写过的方法，没有重写过则调用父类自己的方法**。

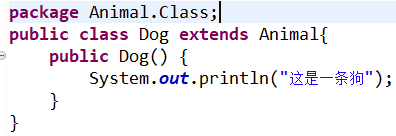
优点：1.减少类与类之间的耦合关系(低耦合) 2.可替换性、可扩展性、灵活性、简化性、接口性。

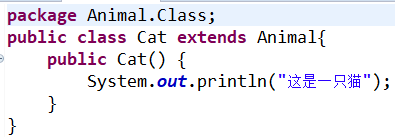


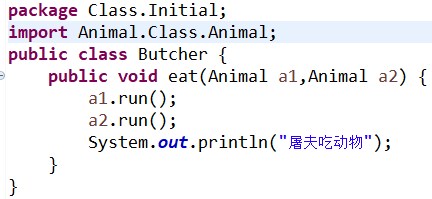


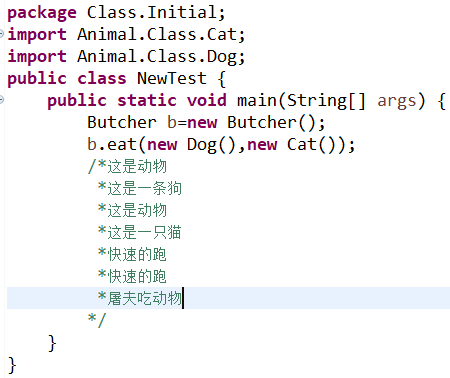
**多态的应用实例：**











## 十四、常量池、栈、堆

### 1.堆(heap)

java堆的存在目的是存放对象实例，几乎所有的对象实例都在这里分配内存，是被所有线程所共享的一块内存。当堆内存不够就会出现OutOfMemoryError异常。new 创建的实例化对象和数组存放在堆中，堆内存中所有的实体都有内存地址，堆内存中的实体是用来封装数据的，这些数据都有默认值。当堆内存中的实体不再被指向是，JVM启动垃圾回收机制，不定期自动清除。

### 2.栈(stack)

Java虚拟机栈，线程是私有的，基本数据类型、局部变量、对象的引用变量都存放在栈中，数据一执行完毕，变量会立即释放，节约内存空间。栈内存中的数据没有默认初始化值，需要手动设置。

注意方法中的局部变量被final修饰之后是存放在堆种，而不是栈种。

**静态变量存放在方法区中**。

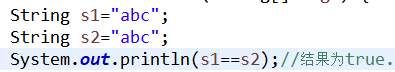
### 3.字符串常量池(String Constant Pool)

JDK1.7之后的版本中，字符串常量池被放在了堆里面；优点是减少相同内容字符串的创建，节约内存。

### 4.字符串的创建方式([四、字符串](#_四、字符串4.字符串的创建方式))

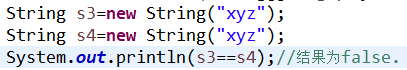
下面这两种解释好像有点争议，到时候自己再细细研究

①采用字面量创建



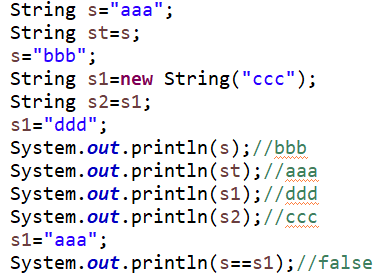
采用字面量创建字符串时，JVM首先在字符串常量池中查找是否有“abc”这个对象，若不存在，则在字符串常量池中创建”abc”这个对象，然后将这个对象的引用地址返回给“abc”对象的引用s1，这样s1会指向池中”abc”这个字符串对象。若存在，则不创建任何对象，直接将“abc”的地址返回并赋值给s2。因为s1和s2都是指向字符串常量池同一个”abc”对象，所以结果为true。

②采用new标准的构造方法创建



采用new方法创建字符串时，JVM先在字符串常量池中查找是否有“xyz”这个对象，如果有，则不在池中再去创建“xyz”这个对象，而是直接在堆中创建一个”xyz”对象，并将堆中的“xyz”这个对象的地址返回赋值给引用s3，这样s3就指向了堆中的”xyz”这个对象。如果字符串常量池中没有“xyz”这个对象，JVM则先在字符串常量池中创建一个”xyz”这个对象，然后再在堆中创建“xyz”这个对象，然后把堆中的”xyz”这个对象的地址返回并赋值给s3，这样s3就指向了堆中的“xyz”这个对象。S4则是向堆中创建了另一个”xyz”对象，由于s3和s4指向的是不同的对象(对象的内容虽然相同，但是引用所指向的地址却不相同)，所以结果为false。

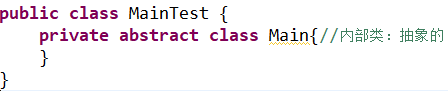
(**下面输出false有点疑问？**)



## 十五、抽象类

**抽象类**：当一个类不足以描述一种事物时，会将其定义为抽象类，用**abstract**修饰。

**抽象类不能被final修饰**。**注意外部类只能用public 、abstract 、final和默认修饰。不能用protected、private、static修饰外部类，但是可以修饰内部类。因此若抽象类作为外部类的话，则不能被private等修饰，若作为内部类的话，则可以被private修饰**。



### 1.抽象类与普通类的区别

①抽象类不能创建对象(即不能用new建对象)，但是有构造器。

②抽象类中可以有抽象方法(用abstract修饰的方法)，但是普通类中没有。

③抽象类中的子类必须重写父类的抽象方法，否则子类报错；或者将子类也变为抽象类。

注意：**抽象方法没有方法体，需要被重写，故抽象方法不能被private、static、final修饰**。



包含一个抽象方法的类必须被声明为抽象类，否则报错。abstract不能用来修饰变量，只能用来修饰类和方法。

## 十六、接口

接口不是类，而是用来描述一种功能或者一种约束规范。用**interface**描述。

**注意接口只能用public 、abstract和默认修饰，不能用protected、private、static、final修饰。和修饰的类差不多，只不过少了一个final。**

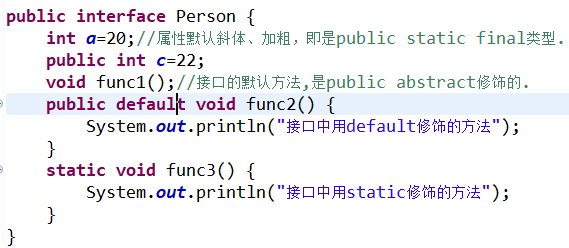
### 1.接口的特点

①接口不是类，所以不能创建对象，没有构造器。

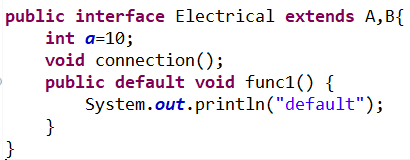
②**接口的默认属性都是public static final修饰的**，且接口的属性只能用public、static 、final和默认修饰，并且接口的属性单个加public、static 、final修饰和不加的效果是一样的，都是public static final类型。即接口中的属性只要写变量类型 变量 变量值即可。

③**接口的普通方法都是public abstract修饰的(即无方法体，且子类必须重写这个方法)**，或者改成default、static修饰的，用这两个修饰的方法要有方法体。

接口中的抽象方法都是用来重写的，而接口中的default或static方法都是接口用来自己写的。

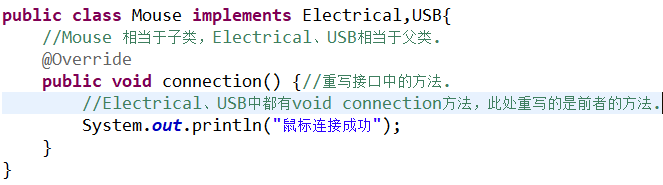


④**接口与接口之间的关系为继承关系**，用extends表示**，可以是多继承。而类与类之间只能是单继承**。

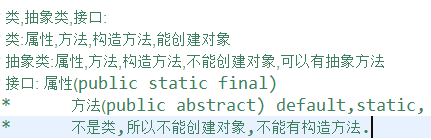


⑤**类可以实现接口，用implements，并且可以是多实现**。其中类相当于子类，接口相当于父类。若在接口中定义了普通方法，则在类中要重写这个方法。

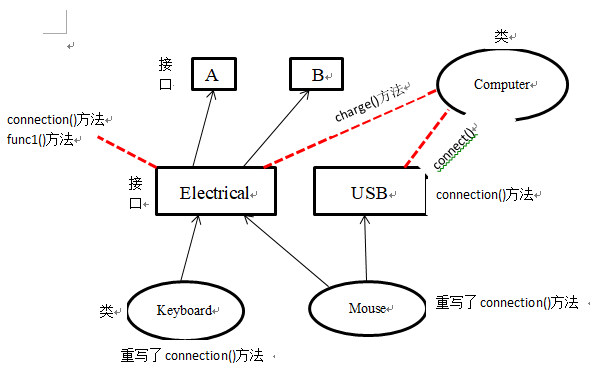
抽象类也可以实现接口，若接口中定义了普通方法，则抽象类的子类要重写这个方法。

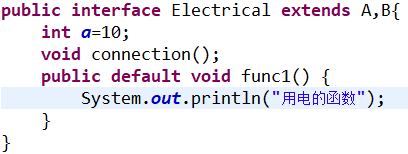
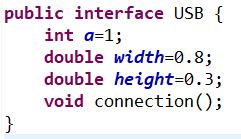


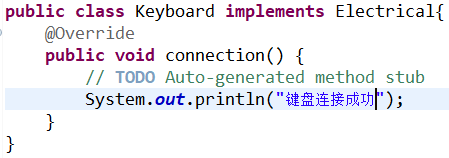
### 2.小结

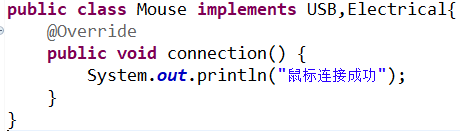


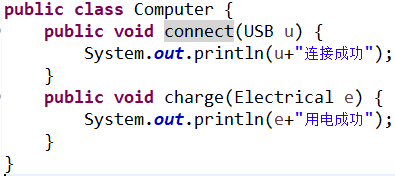
**接口例子：**

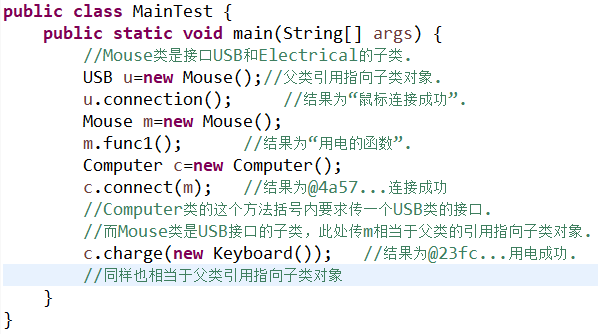


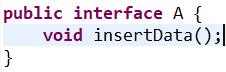


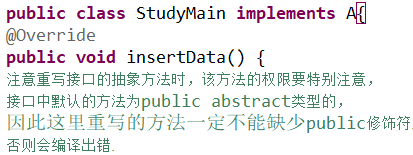












## 十七、成员内部类、静态内部类、局部类、匿名内部类

内部类：有的类不想对外公布，或者公布的权限由某些类决定，此时可将该类变成内部类。

注意在创建类时，类名之后没有小括号()，而是直接大括号{}。**普通内部类是属于外部类的对象，而静态内部类则属于外部类，而不是对象**。

**private、static、protected都不能修饰外部类，但是可以修饰内部类**。

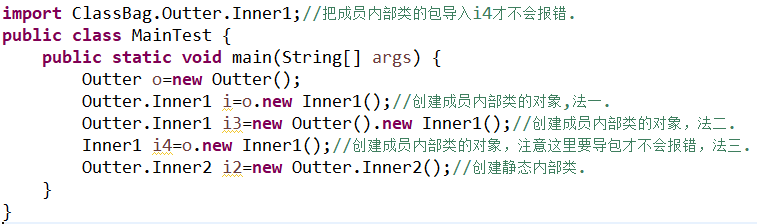
### 1.成员内部类

在类里面再定义一个普通的类。在外部类中要使用普通内部类的属性、方法时，可以在外部类中创建内部类的对象，然后**.属性/方法**。

访问格式：**外部类名.内部类名 对象名=外部类对象.内部类对象**。

### 2.静态内部类

在类里面再定义一个静态的类，**静态内部类只能使用外部类的静态数据**。



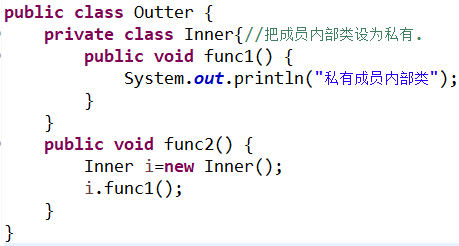
### 3.局部内部类

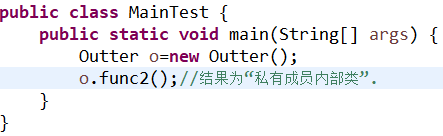
在**类的方法**中再定义一个类。**局部内部类没有权限修饰符，局部内部类只能在该方法里面进行创建对象，在方法中新建的局部变量在其局部内部类中不能修改，在局部内部类中当成是final类型的常量，可以用但不能修改**。

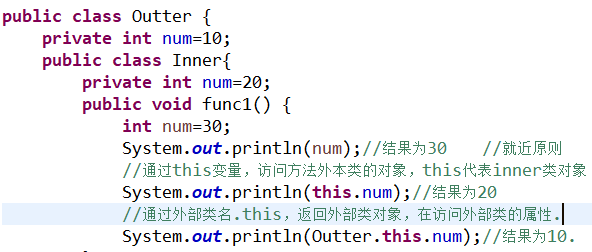
**注意在局部内部类中只能用abstract、final和默认修饰，不能用public、private、protected修饰**。



### 4.成员内部类私有化







### 5.匿名内部类

①**概念**：**内部类的简化写法**

②**前提**：存在一个类(可以是具体或抽象的类)或接口

③**格式**：**new 类名(或接口名){…重写的方法}**

④**本质**：创建的是继承了类或实现了接口的子类匿名对象

⑤**匿名类不能是抽象的，它必须要重写所继承的类或者实现的接口的所有抽象方法。总是一个内部类，且不能被static修饰。匿名类是隐式的final**。

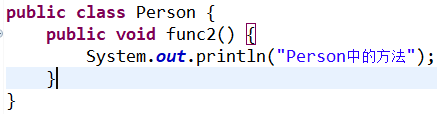
⑥使用匿名内部类时，必须有且仅能继承一个类或实现一个接口。

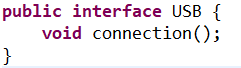
⑦匿名内部类中不能定义构造函数，且不存在任何静态成员变量和静态方法，不能定义静态初始化代码块，不能在匿名类中定义接口。

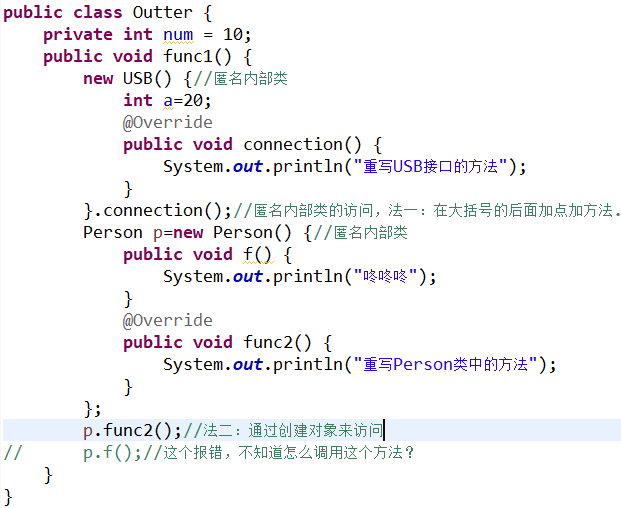
⑧匿名内部类的访问权限：访问外层类中的属性。

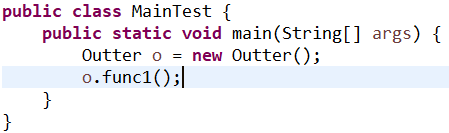
⑨**匿名内部类为局部内部类，**故局部内部类的所有限制同样对匿名内部类生效。匿名内部类可以有属性方法等。

⑩访问方式：









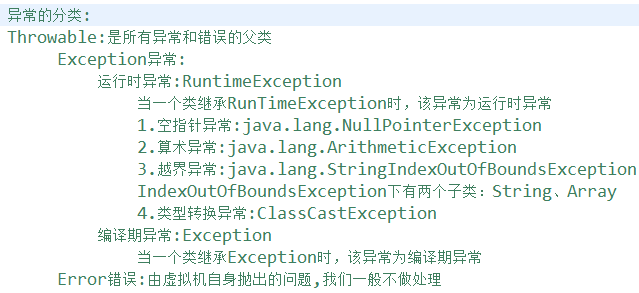
**匿名内部类在开发中的使用，一般是方法参数为接口对象**：



## 十八、异常

**异常**：中断正常指令流的一种事件，可处理。

### 1.异常分类

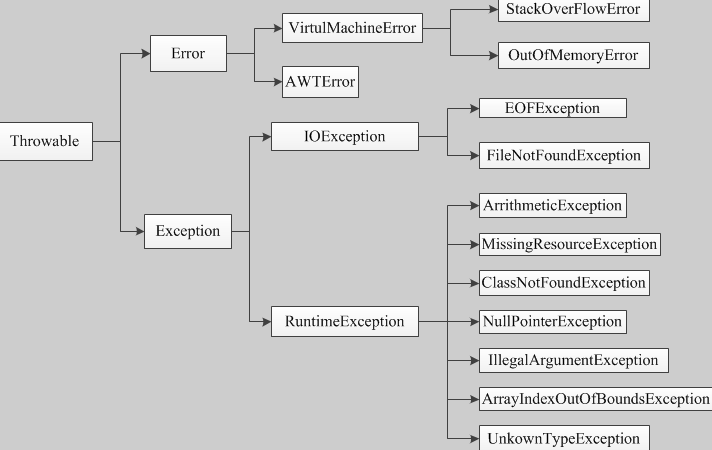


Error错误是由java虚拟机生成并抛出，大多数错误与代码编写者所执行的操作无关。

常见的**错误**：StackOverFlowError、OutOfMemoryError。

常见的**编译期异常**：ClassNotFoundException、FileNotFoundException、SQLException、IOException。

常见的**运行期异常**：NullPointerException、IndexOutOfBoundException、ArithmeticException、ClassCastException。



java中的异常也可分为**检查异常**和**非检查异常**。**检查异常包括编译期异常，非检查异常就包括运行时异常和Error**。

**RuntimeException**中time的t是小写，运行时异常中还有：

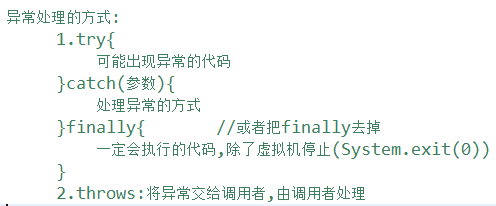
类型不匹配异常：**java.util.InputMismatchException，**注意这个异常还要导包。而上面写的4个异常不用导包。

类的全名称：是类名+包名；java.lang.NullException中的java.lang是包名。System.exit(0)是终止java虚拟机的。

异常有的在java.lang包中，有的在java.util中。异常是写在方法中的。

### 2.异常处理方式

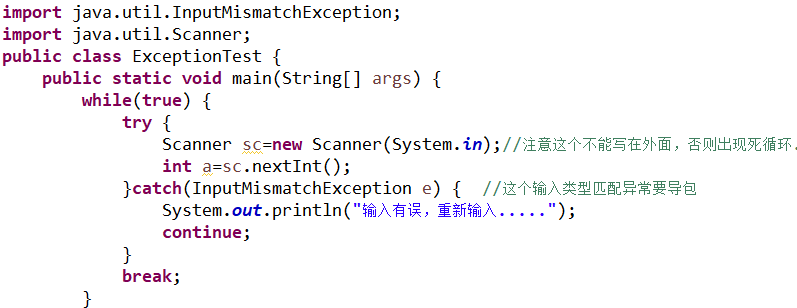
使用多重的catch捕获异常时，异常的顺序：**先子类异常再父类异常**。**可以加一个catch(Exception e)用来处理可能遗漏的异常**。



有多个异常时，用或**|**，不能用两个**||**，否则报错。或者再写一个catch语句。



finally语句中的一定会执行的代码是说：不管上面的代码是否有错，一定会执行finally语句中的内容。



**处理异常的选择**：try..catch还是抛出异常

**①**在MVC[MVC介绍](../4.javaEE基础/8.mvc.docx)框架中，Controller层的异常必须捕获。一般不允许将系统内部的异常不做任何封装处理直接抛给客户端，这样对系统来说会暴露过多信息。

**②注意throws抛出异常之后，后面的代码就不会继续执行了，而try...catch时，捕获到了异常之后，后面的代码还是会继续执行**。

### 3. try catch finally中return的执行情况

try语句在执行return语句之前，将其他的所有操作执行完，保留好要返回的值，而后转入执行finally中的语句。分如下三种情况：

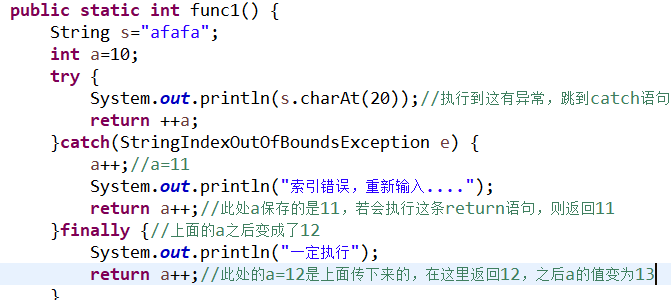
**⑴**若finally中有return语句，则会将try中的return语句“覆盖”掉，直接执行finally中的return语句，得到返回值。

**⑵**若finally中没有return语句，且没有改变返回值，则执行完finally中的语句之后，接着执行try中的return语句，返回原来保存的值。

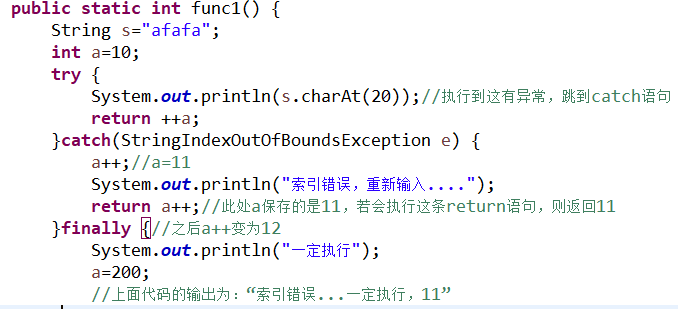
**⑶**若finally中没有return语句，但是改变了要返回的值，可分为以下两种情况：

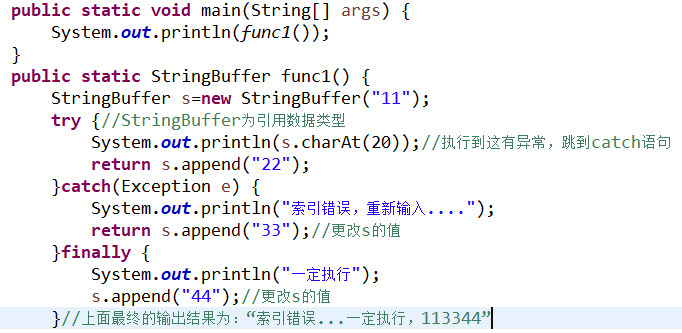
**①**若return的数据是基本数据类型或文本字符串，则在finally中对该基本数据类型的改变不起作用，try中的return语句返回的是在进入finally语句之前保存的值。

**②**若return的数据是引用数据类型，在finally中对该引用数据类型的属性值的改变起作用，try中的return语句返回的就是finally中改变后的该属性的值。



当上面finally语句中没有return语句时，执行的结果为：“索引错误....”，之后为“一定执行”，之后返回“11”，然后a的值变为11+1即12。这个顺序要特别注意。

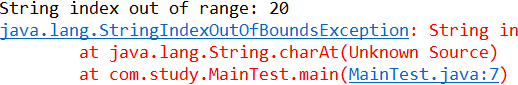


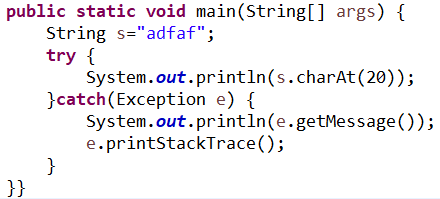


### 4.Exception e的方法

**e.getMessage()：**以字符串的形式返回异常或错误的详细信息。

**e.printStackTrace()：**将异常或错误信息以红色字体的形式打印出来，并可以查看异常或错误的位置。





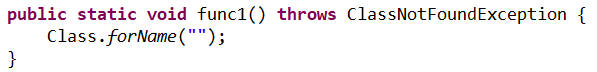
### 5.throw、throws、throwable的区别

**⑴throw：**用在方法体内，表抛出异常，由方法体内的语句处理，**throw抛出的是一个异常对象**，try...catch中catch的参数接受的就是throw关键字抛出的对象。**throw 后面接的是异常对象，或异常对象的引用**。





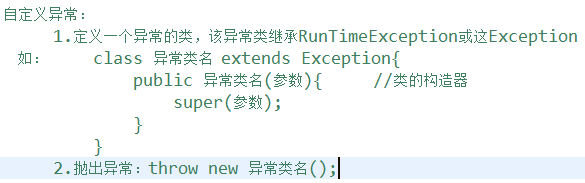
**⑵throws：**用在方法声明的后面，表该方法可能要抛出异常，且抛出的异常由该方法的调用者处理，若有多个异常则用逗号隔开。格式如下：



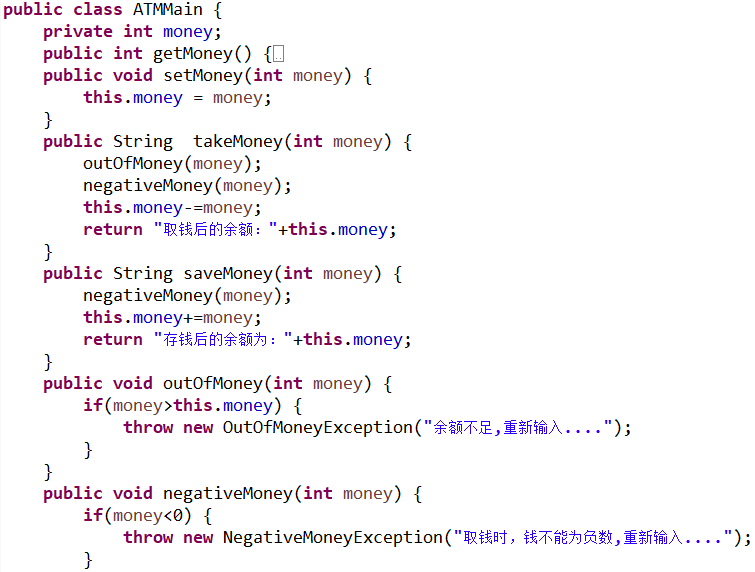
**⑶throwable：**是所有错误和异常的父类。

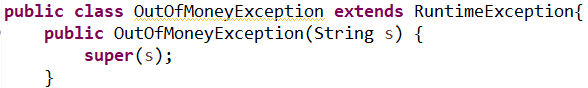
### 6.自定义异常

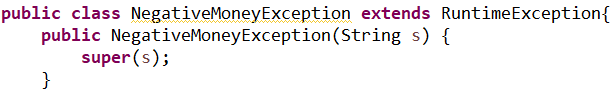
在自定义异常时用throw，且要将定义的异常类继承父类异常(运行时异常继承RunTimeException，编译时异常继承Exception)。

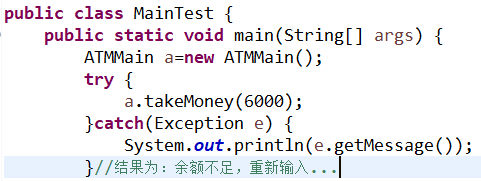


**自定义异常的例子：**









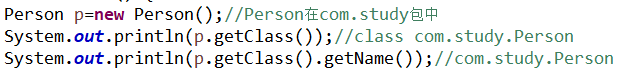
## 十九、内容补充

在做项目时，在一个包中调用另一个包中的方法时，该方法必须是public类型，且若该方法是static类型，则调用该方法时必须用**类名.方法名**。在一个包中调用另一个包中的public static属性时，也是一样的，用**类名.属性名**。

**Java核心思想**：高内聚、低耦合。耦合：类与类之间的关系。

Object**.getClass()**：返回这个object对象的一个Class对象。

Class**.getName()**：返回这个Class对象的全名称(包名+类名)。



斜体：是static类型；加粗：是final修饰的常量。Interface接口的属性是斜体加粗的，即是static final类型。

**java三大特性**：**封装、继承、多态**。

抽象类中可以有普通方法，可以拿来直接用，抽象方法重写再用。

接口中的普通方法就是抽象的方法，不能直接拿来用，要重写方法，才能用。

**魔数**

**异常中弄清楚Scanner sc=new Scanner(System.in)在while循环外面，而int a=sc.nextInt()在循环里面而出现的死循环问题。**

**.class文件中的具体内容**

**String的源码**

**构造块**