第六章 面向对象基础

6.1认识类与对象

Java语言是一种面向对象的语言；面向对象的三大特征：继承、封装、多态。

6.1.1 属性

对象具有各种特征。

6.1.2 方法

对象的行为。

6.1.3 对象

用来描述客观事物的一个实体，由一组属性和方法构成。

6.1.4 类

具有相同属性和方法的一组对象的集合；

类是对象的抽象，对象是类的具体；

可以认为类是一个模板，通过给模板加入数据，就生成了一个一个对象。

6.1.5 创建对象

类名+对象名=new 类名（）；

如：Perso p=new Person（）；

使用new关键字，创建对象，这个过程叫实例化。

如果初始化对象，没有赋值，其默认值规则如表6.1：

**表6.1：默认值规则**

|  |  |
| --- | --- |
| 数值类型 | 默认值 |
| byte,short,int,long | 0 |
| flaot,double | 0.0 |
| boolean | false |
| char | /u0000 |
| 引用类型 | null |

6.1.5 使用对象

对象名.属性；

对象名.方法（）；

6.1.6 对象数组

对象数组的数据类型就是具体的类名；

对象数组存储的就是这个类的对象；

每个元素就是一个对象；

可以利用下标找到相应的对象；

可以按照使用对象的方法使用该元素。



图6.1 数组对象内存管理

6.2 方法及方法重载

6.2.1 类的三个部分

成员变量，成员方法，内部类。

6.2.2、成员变量：

直接定义在类中的变量，定义于方法外部。不赋值时，默认值规则如表6.2：

**表6.2：默认值规则**

|  |  |
| --- | --- |
| 数值类型 | 默认值 |
| byte,short,int,long | 0 |
| flaot,double | 0.0 |
| boolean | false |
| char | /u0000 |
| 引用类型 | null |

6.2.3、成员方法：

定义在类中的方法。

6.2.4、形参与实参

形参——定义在方法内的参数；

实参——调用方法时传入的参数。

6.2.5、成员变量与局部变量

成员变量——直接定义在类中的变量；

局部变量——定义在方法中的变量。

区别：

1、作用域不同

局部变量只作用于方法内部，该方法外 无法访问；成员变量作用于类中，该类中的所有成员方法都可以调用；如果权限允许，甚至可以作用与类外部。

2、初始值不同

成员变量如果不赋值，会拥有默认的值；局部变量必须先初始化再使用。

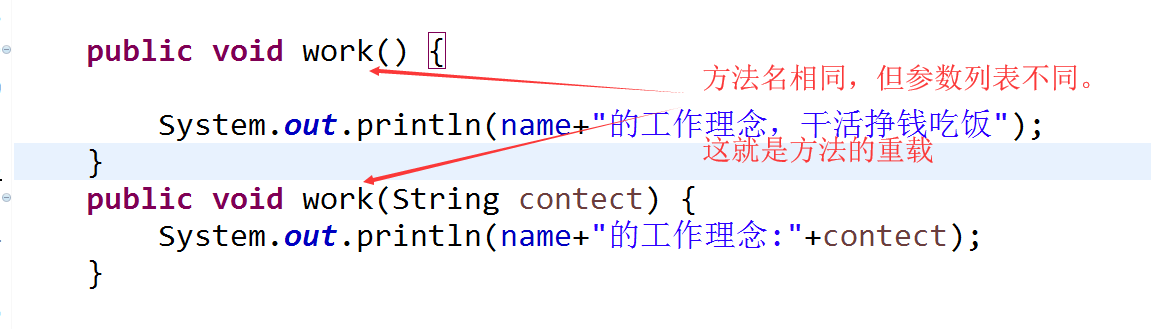
3、同一方法中不允许出现同名的局部变量；在不同的方法中可以有同名的局部变量名

4、局部变量名可以和成员变量名同名，且成员变量的优先级更高。

6.2.6 方法重载

一个类中定义多个同名的方法，但每个方法具有不同的参数类型或个数。

方法的签名：方法名+参数列表。



6.2.7 方法重载的特点

·在同一个类中；

·方法名相同；

·参数的个数或类型不同；

·方法的返回值不能作为判断方法之间是否构成重载的依据；

·方法的修饰符(public/private...) 不能作为判断方法之间构成重载的依据；

* 构成方法重载后，根据所需要的返回类型和参数列表选择所要使用的方法，方法重载简化了调用方法的代码。

6.2.8 构造方法

当类创建一个对象时，会自动调用该类的构造方法；构造方法分为默认构造方法和带参数的构造方法。

1、构造方法的定义

构造方法没有返回值，也不能用void表示；构造方法名与类名一致。其中默认构造方法不带参数。

当没有定义构造方法时，系统会自动添加一个默认构造方法。

6.3 this的使用

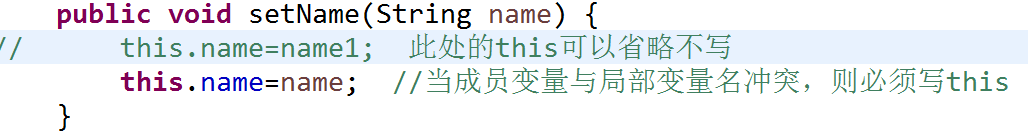
一个对象的默认引用。每个实例方法内部都有一个this引用变量，指向调用 这个方法的对象。

this:指代当前对象

\* 对一个对象的默认引用。

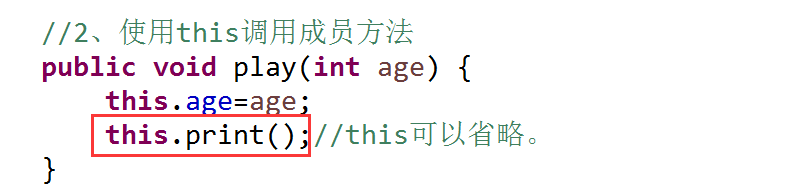
\* 每个实例方法内部都有一个this引用变量，指向调用这个方法的对象

6.3.1、使用this解决成员变量与局部变量同名冲突的问题

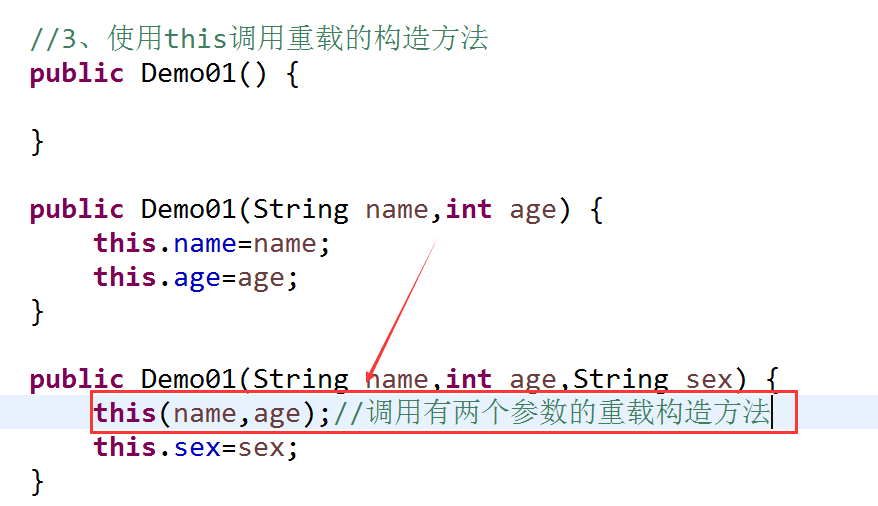


在方法中只要涉及成员变量，都应该加this，只不过当名字不冲突时，可以省略。

6.3.2、使用this调用成员方法,this可以省略



6.3.3、this可以调用重载的构造方法

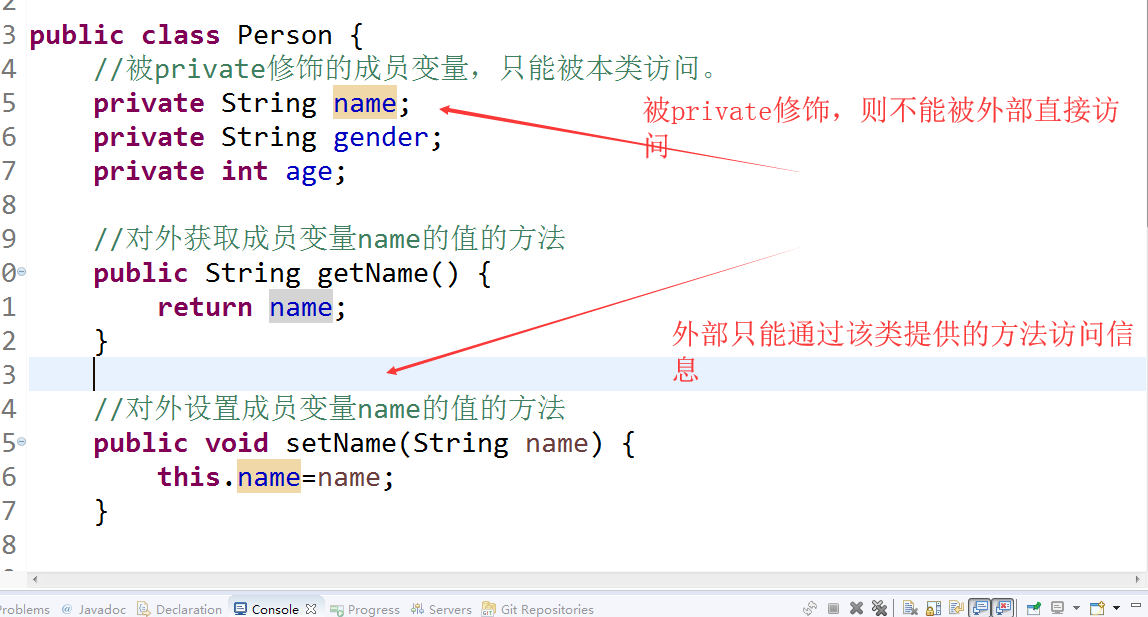


注意：只能在构造方法中使用，且必须放在最前面

6.4 面向对象三大特征之一：封装

其实质是将类的状态，信息隐藏在类内部，不允许外部程序直接访问。 而是通过该类提供的方法实现对隐藏信息的操作和访问。

如：



输出结果：



6.5 package

6.5.1 包

package 必须是java源文件的第一条非注释语句。且一个源文件只有一个包声明语句。

编码规范：

1、一个唯一的包名前缀通常是全部小写的ASCII字母(abcd.....)。并且是一个顶级域名com,edu,gov,net,org。通常使用组织的网络域名逆序。

比如：域名为：javagroup.net

包声明： package net.javagroup.mypackage

2、包名后续部分，依据机构内部而决定。

比如： net.javagroup.reseach.powerproject

↑ ↑

项目名 部门名

\* 北大青鸟网址： bdqn.cn 项目名：itrip 部门：tch

\* 包命名： cn.bdqn.itrip.tch 按照规范，这样可以保证包名不重复。

6.5.2包的作用：

1、包允许将类组合成较小的单元，易于找到和使用相应的类文件。

2、防止命名冲突：Java中只有在不同的包中的类才能重名。

3、包允许在更广的范围内保护类、数据、方法

6.5.3 包的导入——import

实例化Scanner，应该要如下写法：

java.util.Scanner sca=new java.util.Scanner(System.in);

但这么写太麻烦了。

所以可以先导入: **import** java.util.Scanner;

然后就可以直接如下写书：

Scanner input=new Scanner(System.in);

6.6 访问修饰符

6.6.1 类的访问修饰符

|  |  |  |
| --- | --- | --- |
| 作用域  修饰符 | 同一包中 | 非同一包中 |
| public | 可以使用 | 可以使用 |
| 默认修饰符（不写） | 可以使用 | 不可以使用 |

6.6.2 类成员的访问修饰符

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 作用域  修饰符 | 同一类中 | 同一包中 | 子类中 | 其它地方 |
| Private | 可以使用 | 不可以使用 | 不可以使用 | 不可以使用 |
| 默认修饰符（不写） | 可以使用 | 可以使用 | 不可以使用 | 不可以使用 |
| Protected | 可以使用 | 可以使用 | 可以使用 | 不可以使用 |
| Public | 可以使用 | 可以使用 | 可以使用 | 可以使用 |

6.6.3 static 关键词

static三种修饰

1、static修饰成员属性：静态变量（类变量），可以直接使用类名调用静态变量

\*被static修饰的变量

\*在内存的静态区/方法区中仅有一个拷贝

\*类内部中，可以在任何方法直接访问静态变量

\*其他类中，也可以直接通过类名访问

2、static修饰方法：称为静态方法/类方法

\*在静态方法中不能直接访问实例变量、实例方法

\*在实例方法中能够调用静态方法 静态变量

3、static块：

\*如果有多个静态块，按顺序加载

\*每个静态代码块只会被加载一次

\*static块属于类的代码块，在类的加载期间执行代码块，只执行一次，可以用来在软件中加载静态资源

\*static块常常用来加载静态资源（图片、音频、视频......）

6.6.4 final 关键词

final的三种用法

1、final修饰成员变量和局部变量

1.1、final修饰成员变量

\*意为不可改变

\*两种初始化方法：1、声明的同时初始化；2、构造方法初始化；

1.2、final修饰局部变量

\*意为不可改变

\*在使用之前初始化

2、final修饰方法：不可以被子类重写。

\*其意义为：防止子类定义新方法时，不经意将该方法重写掉

3、final修饰类：意为不可继承。

\*意义在于：保护类不被继承，控制滥用继承对系统造成危害

\*比如String类就是被final修饰

6.6.5 static final 修饰的成员变量

\*不能改变，并且唯一

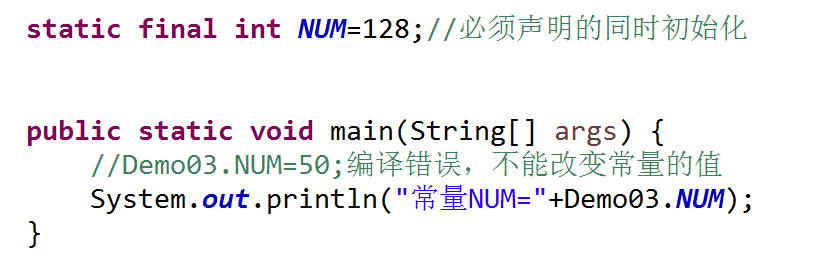
\*必须声明的同时初始化

\*常量的名称建议全部大写

\*在编译期时，被直接替换为具体的值。效率高

\*可以用类名去调用常量

\*在方法区里面，只有一份，不可更改



6.7 内存

内存管理： 由JVM来管理

编译好的java程序需要运行在JVM中。

代码还是数据，都需要存储在内存中，JVM为java程序提供并管理所需要的内存。

JVM内存分为：堆，栈，方法区。分别存储不同的数据

6.7.1堆：

1用于存放new出来的对象（包括成员变量）。

2成员变量的生命周期: 通过new创建对象时存在堆中，对象被垃圾回收器GC回收时 一并消失。

3堆中的垃圾： 没有任何引用所指向的对象。

垃圾回收器（GC）不定时的到堆中查看，看到垃圾就自动回收。当然这个回收的过程是透明，并不是一有垃圾就立刻回收。 如果想快点回收垃圾，可以调用System.gc().

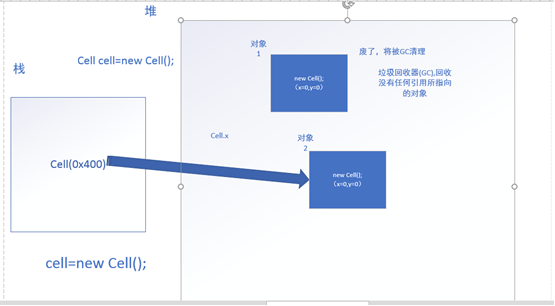
当然这也是建议快点回收垃圾，也不是立马回收。

垃圾回收器：是JVM自带的一个线程。（自动运行的程序），用于回收任何没有引用指向的对象。

因为有垃圾回收器的存在，所以java程序员不用更担心内存管理。

内存泄漏：不在使用的内存，没有被垃圾回收器及时回收。

严重的内存泄漏:会因为过多的内存占用而导致程序崩溃。



当堆中某个对象没有引用，则会被垃圾回收器(GC)，清理

6.7.2栈：

1、用于存放正在调用方法中的所有局部变量（包括参数）。

2、调用方法时，在栈中为该方法分配 了一块对应的栈桢，栈桢中包含了所有的局部变量（包含参数）。

方法结束后，栈桢消失，局部变量一并消失。

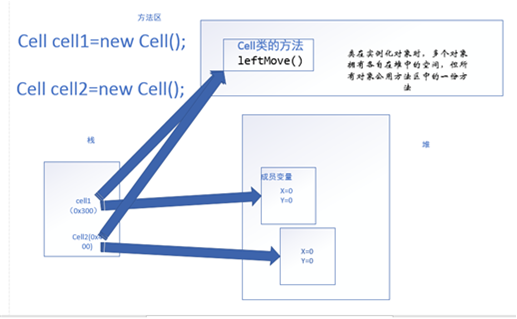
3、局部变量的生命周期： 调用方法时存在栈桢中，方法调用结束时与栈桢一并消失。

6.7.3方法区/静态区：

1用于存放.clss字节码文件，其中包含了方法。

java程序运行时，首先通过类装置载入字节码文件，经过解析后将其装入方法区，里面有类的各种信息，包含了方法。

2方法只有一份，通过this来区分具体是哪个引用调用。( 类在实例化对象时，多个对象拥有各自在堆中的空间，但所有对象公用方法区中的一份方法。)



第七章 继承和多态

7.1 继承

7.1.1 继承的概念

新类在不增加自身代码的情况下，通过从现有的类中继承其属性和方法，来充实自身内容的这种现象称为继承。

1、可继承内容：

·public、protected修饰的属性和方法，可以跨包继承；

·默认修饰符符修饰的属性、方法，不可跨包；

·无法继承父类的构造方法；

2、特性：

·子类拥有父类非private的属性，方法；

·子类可以拥有自己的属性和方法，可以对父类进行扩展；

·子类可以用自己的方式实现父类的方法；

·Java的继承是单继承，但是可以多重继承 ；

·提高了类之间的耦合性；

·继承使代码可以重用，增加软件的可扩充性。

7.1.2 super的用法

supper的三种用法

1、访问父类属性

2、访问父类方法

3、访问父类构造方法

注意：

1、使用super关键字,super代表父类对象

2、在子类构造方法中调用且必须是第一句

3、不可以访问父类中定义为private的属性和方法

7.1.3 继承条件下构造方法的调用规则

1、子类没有通过super显式调用父类有参构造方法，且没有通过this调用自身构造方法，系统会默认调用父类无参构造方法；

2、如果子类使用super调用了父类有参构造方法，将执行对应的有参构造方法；

3、如果存在多级继承关系，则在实例化一个子类对象时，上述规则将向更高级传递，直到父类为Object；

7.1.4 方法重写

子类根据需求对从父类继承的方法进行重新编写；

要求：

1、方法名相同、参数列表相同；

2、返回值类型相同，或者是其子类；

3、重写的方法的访问权限不能缩小；

4、父类是静态方法，重写的方法也必须是静态方法；

5、子类不能抛出比父类更多的异常；

方法重载与方法重写的区别

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 修饰符 | 方法名 | 参数列表 | 返回值类型 | 范围 |
| 重载 | 无关 | 相同 | 不同 | 无关 | 同一个类 |
| 重写 | 不能更严 | 相同 | 相同 | 相同 | 子类 |

7.2 多态

7.2.1 多态的概念

同一个引用类型，使用不同的实例而执行不同的操作， 呈现出不同的结果；

多态依赖重写实现。

7.2.2 向上转型

子类向父类转型称为向上转型。

格式：

父类型 引用变量名=new 子类型()；

规则：

1、向上转型时，子类型自动转为父类型；

2、通过父类引用变量调用子类重写的方法，而不是父类的方法；

3、子类特有的方法，不能通过父类引用变量调用。

7.2.3 向下转型

将父类对象赋值给子类，即父类转换为子类（需要强制转换）。

格式：

子类型 引用变量=(子类型) (父类型引用变量)；

instanceof：

解决了在向下转型的过程中，如果不是转换为真实的子类，则编译错误，可以使用instanceof关键词，实现预判下是否能转换为其子类。

语法：父类引用变量 instanceof 子类型

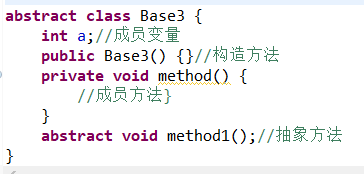
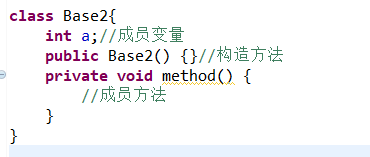
第八章 抽象类和接口

8.1 抽象类

8.1.1 抽象类的定义

当一个类的方法被abstract修饰时，该方法称为抽象方法，抽象方法所在的类被称为抽象类，抽象类也必须用abstract修饰。

8.1.2 抽象类与普通类的区别

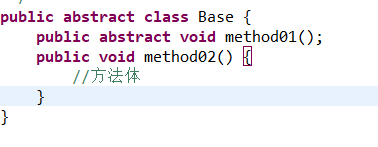
**图8.1 抽象类和普通类**

·抽象类必须用abstract修饰，普通类不需要；

·抽象类不能被实例化，普通类可以实例化；

·抽象类可以拥有普通类的所有成员，普通类不能有抽象方法；

8.1.3 抽象方法与普通方法的区别



抽象方法

普通方法

·抽象方法需要用abstract修饰，普通方法则不需要；

·抽象方法没有方法体，普通方法有方法体；

8.1.4 抽象类的应用

·抽象类中已经实现的方法可以被子类使用，使代码可以重用；

·抽象方法保证了子类具有自身的独特性；

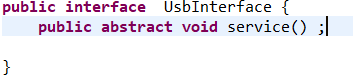
·不能直接实例化抽象类，但是它的非抽象子类可以实例化，也可以通过向上转型实现实例化；

·子类通过extends关键字继承一个抽象类，必须重新父类中的所有抽象方法；

8.2 接口

8.2.1 接口的定义

接口可以认为是一种特殊的抽象类，使用interface定义，权限修饰符只能是public；



8.2.2 接口和的特点

·定义接口使用interface修饰，实现接口使用implement关键词；

**·接口中可以定义常量，不能定义变量，接口的属性默认使用public static final修饰，且必须定义的同时初始化；**

·接口中所有的方法都是抽象方法，默认使用public修饰；

**·接口不能实例化，不能有构造方法（抽象类有构造方法）；**

·接口之间可以通过extends实现继承，一个接口可以继承多个接口，但接口不能继承类；

·**一个普通类只能继承一个父类，但是可以实现多个接口**，并且这个类必须实现接口的所有方法，继承多个接口时，extends必须位于implement前；

·接口不能实现接口；

8.2.3 面向对象设计的原则

1、摘取代码中变化的行为，形成接口；

2、多用组合，少用继承；减少代码的冗余。

3、针对接口编程，不依赖于具体实现；

尽量依赖接口，尽可能少依赖子类，接口相当于子类更加稳定。

4、针对扩展开放，针对改变关闭

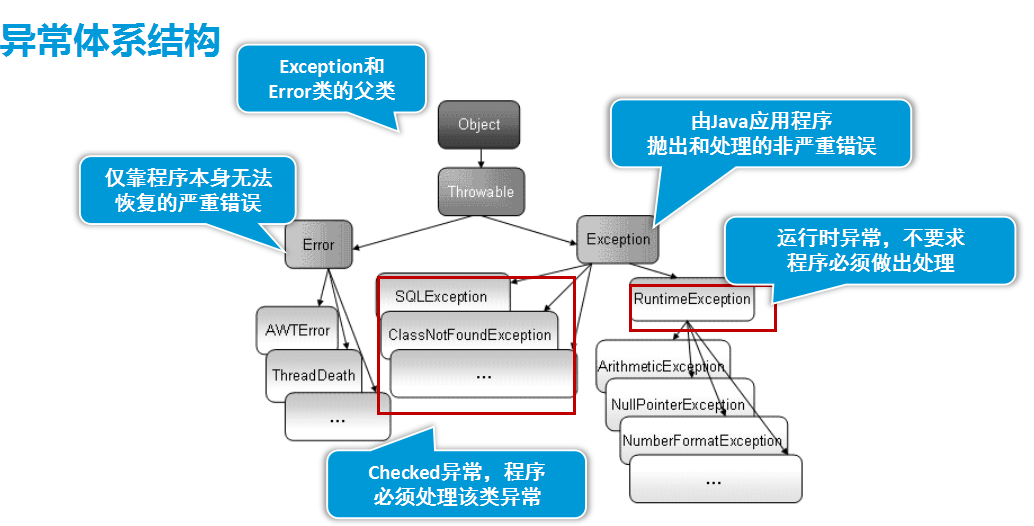
项目的需求发生了变化，应添加一个新的类或接口，不去修改原有代码。

第九章 异常

9.1 异常的定义

异常是程序运行过程中发生的不正常的事件，如下标越界、算术运算出错、对null对象操作……

9.1.1 异常的体系结构

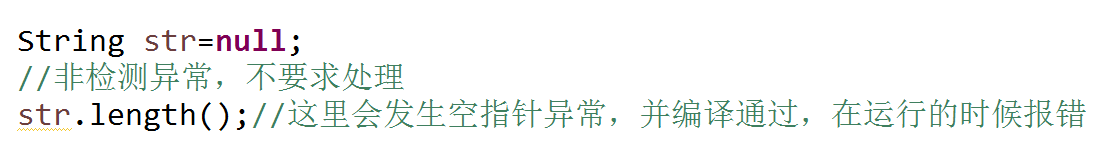


Throwable:所有的异常都是Throwable的子类，派生出Error类和Exception类

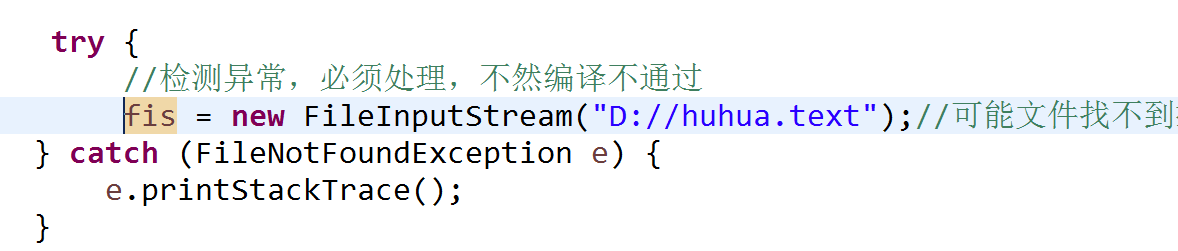
Error类：仅靠程序本身无法恢复的严重错误，比如：内存溢出，动态链接失败，虚拟机错误。一般由虚拟机抛出，只能避免这种异常。

Exception类：由java应用程序抛出和处理的非严重异常，比如:数组下标越界，空指针异常，类型转换错误……Exception类又分为两大类：

①非检测异常(运行异常)：**编译期无异常，运行时出现异常**，不要求程序员必须处理，如：



②检测异常(非运行异常)：**编译期就不能通过**，必须要求程序员处理，否则无法正常运行，如：



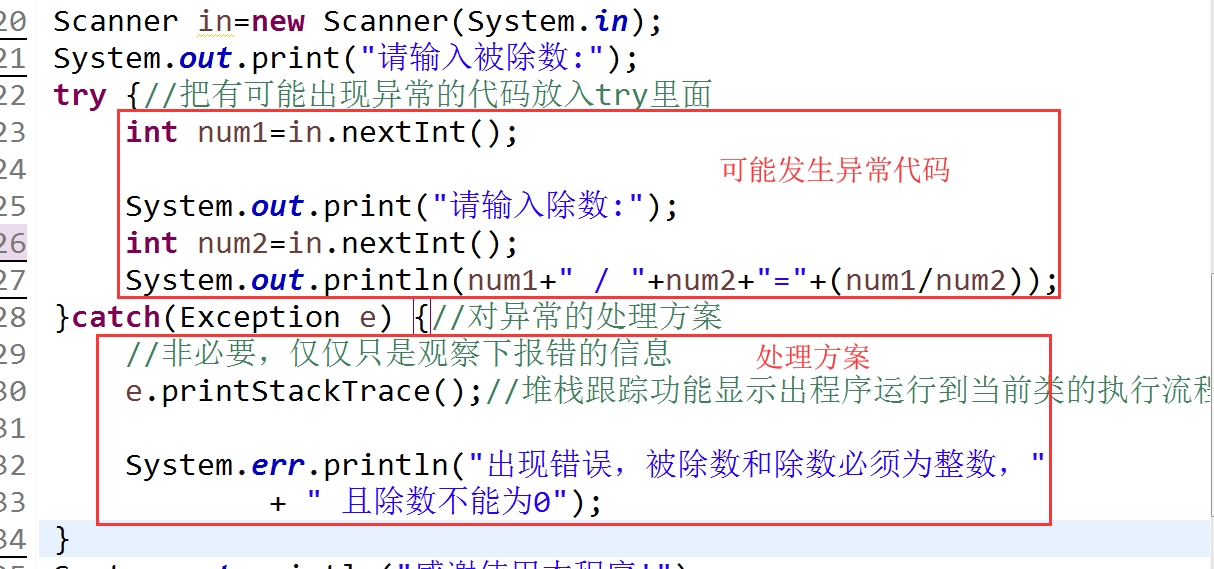
9.1.2 常见的异常

|  |  |
| --- | --- |
| 异常 | 说明 |
| Exception | 异常层次的根类 |
| ArithmeticException | 算术错误异常，如除数为0 |
| ArrayIndexOutOfBoundsException | 数组下标越界 |
| NullPointerException | 访问null对象成员 |
| ClassNotFoundException | 不能加载所需要的类 |
| InputMismatchException | 输入的数据类型与接收类型不匹配 |
| IlleagalArgumentException | 方法收到非法参数 |
| ClassCastException | 对象强制转换出错 |
| NumberFormatException | 数字格式转换异常 |

9.2 异常处理机制

9.2.1 try-catch结构

结构形式：



printStackTrace()：打印异常的堆栈信息 ；

getMessage()：获取异常信息的描述字符串；

try-catch的三种情况：

1、当try中没有发生异常时，catch语句块中的代码都将被忽略；

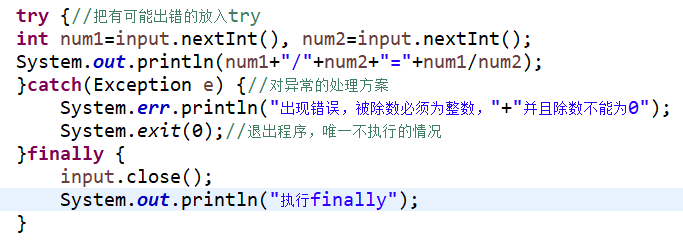
2、当try中发生异常且与catch语句块声明的异常类型相匹配时(类型一致或是它的父类)，try中发生异常的那条代码后面的语句将会被忽略，而相应的catch语句块将会执行；

3、当try中发生异常且与catch语句块声明的异常类型不匹配时，try中发生异常的那条代码后面的语句将会被忽略，且 catch语句块也不会执行；

9.2.2 try-catch-finally结构

结构形式：

try是必须的，catch和finally至少出现其中之一。

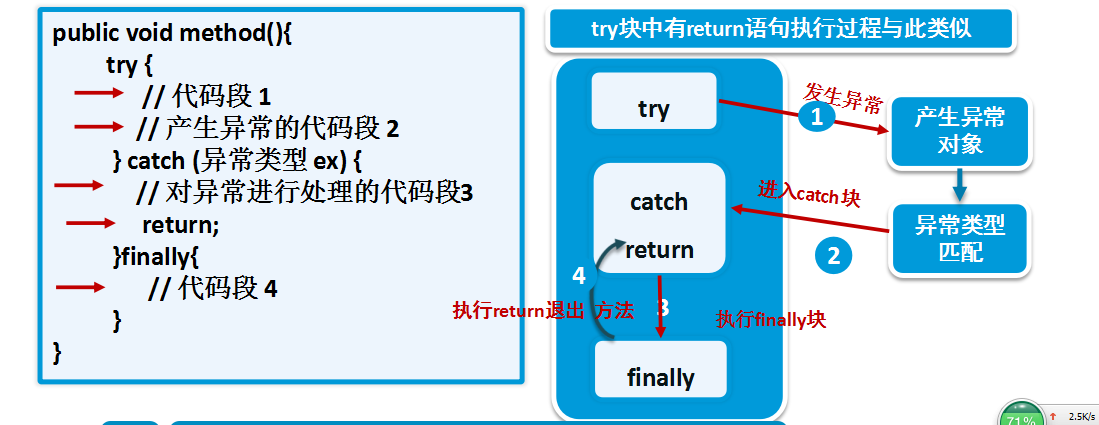


try-catch-finally的4种情况：

1、无异常，try语句块正常执行，catch语句块被忽略，finally语句块也会被执行；

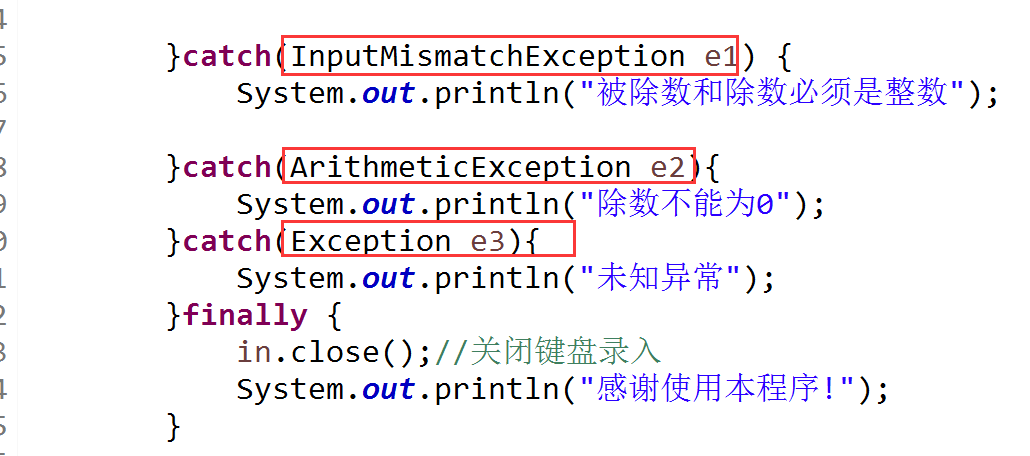
2、发生异常，try语句块后面的代码被忽略，catch语句块捕捉到异常则执行，否则被忽略，finally始终都能执行；

3、当处理异常代码中执行了System.exit(1)，直接退出JVM，finally语句块不执行，也是finally语句块唯一不会执行的情况；

4、当catch语句块中有return时，先执行catch语句块return之前的代码，然后执行finally语句块代码，最后执行catch语句块中return语句退出，具体如下图所示：

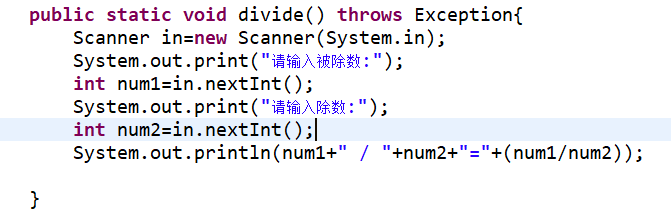
9.2.2 多重catch

当有多重异常需要处理时，可以使用多重catch语句块分别捕捉并处理；catch语句块使用时，顺序必须是从子类到父类，最后一个一般为Exception。



9.2.3 使用throws声明异常

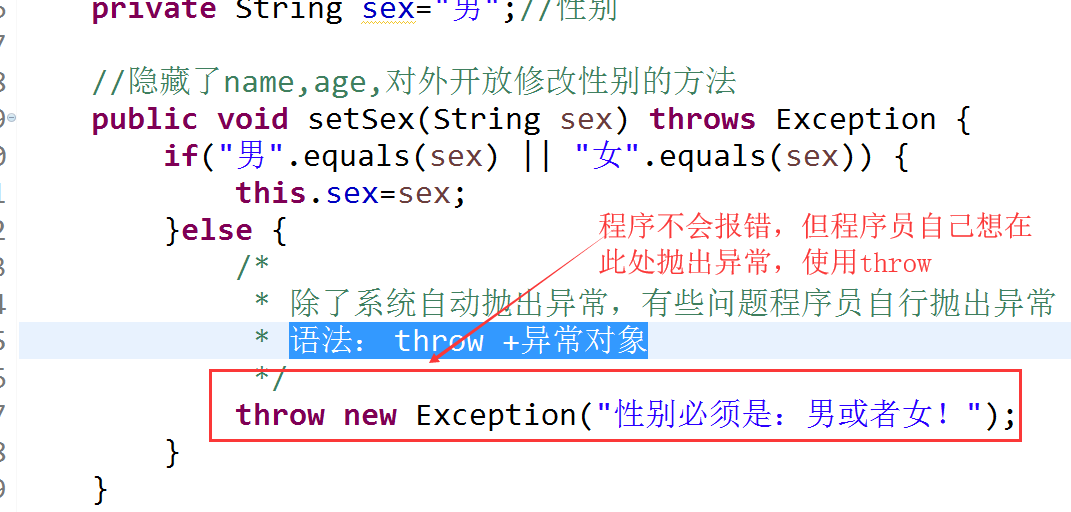
throws：提前声明可能存在的异常；可以声明多种异常，用逗号隔开。如果对异常不做处理，可以继续使用throws声明异常，让上一级处理。main方法声明的异常不做处理则会交给JVM处理（直接中断程序）。



throws声明异常可以不做处理

9.2.4 使用throw 抛出异常

throw： 程序员自行抛出系统无法自动发现、解决的异常；



**throw与throws的区别**

|  |  |  |
| --- | --- | --- |
|  | Throw | Throws |
| 作用 | 程序员自行生成、抛出异常 | 声明该方法内抛出了异常 |
| 使用位置 | 方法体内部，可作为单独语句 | 必须跟在参数列表后面，不能单独使用 |
| 内容 | 只能抛出一个异常对象 | 抛出异常类，可以是多个 |

9.2.5 自定义异常

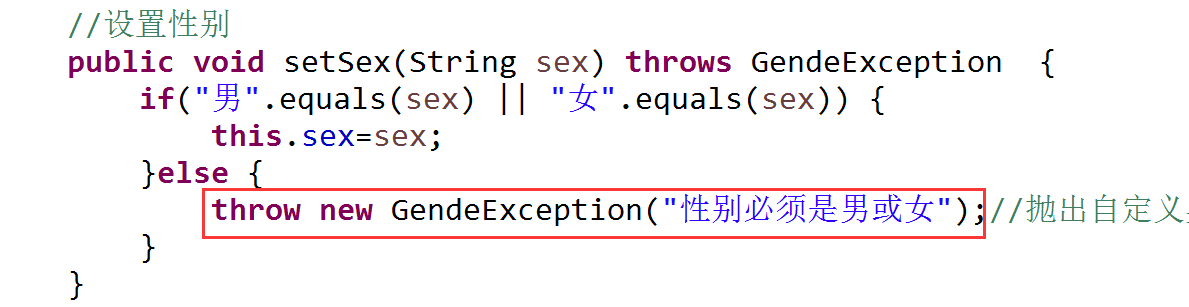


**继承 异常类**

**重写父类构造方法**

使用自定义的异常：

先使用声明异常类，然后在方法体中抛出自定义的异常



声明异常类