**函数的输出，要在代码中打出，不然过了几天就忘了。。。。**

# java语言简介

## java基本概念

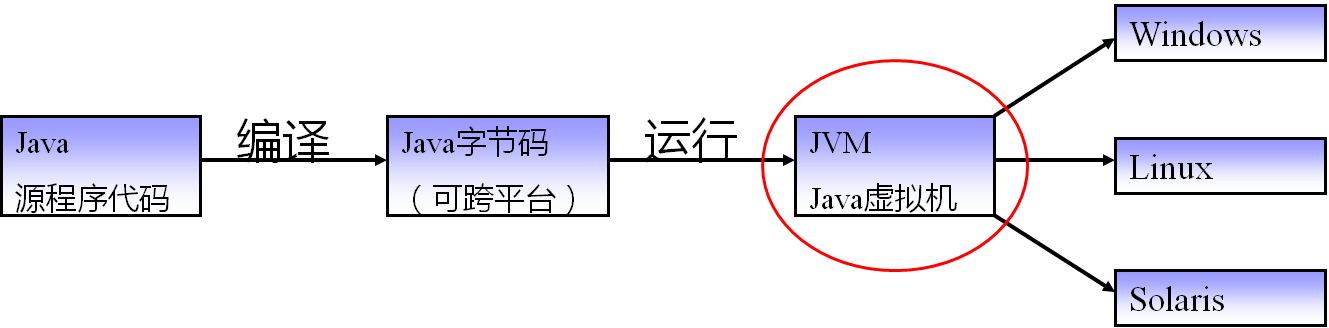
**JDK**：Java Development Kit （ Java 开发必备）

**JRE**：Java Runtime Environment （ Java 执行环境）

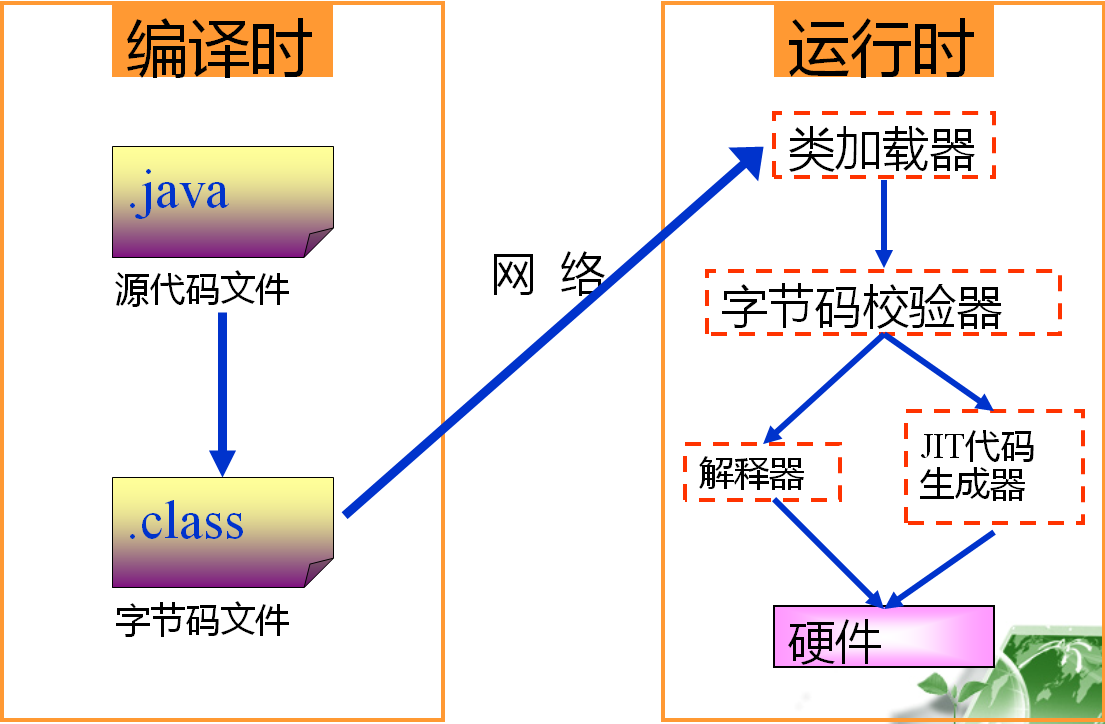
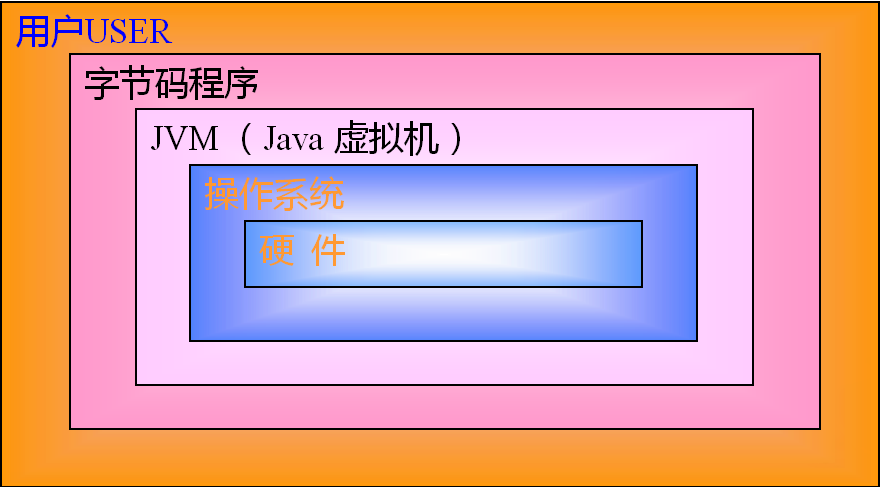
## Java具有平台无关性

Java是一种既面向对象又可以跨平台的语言，即：编写一次，随处运行；

Java不仅适用于单机应用程序和基于网络的程序，也可用于创建消费类设备的附件程序，如移动电话、掌上设备等。



Java虚拟机（Java Virtual Machine）是可运行Java字节码的虚拟计算机系统；使用Java语言编写的程序，实际上是运行在JVM之上，而不是运行在操作系统上；它有一个解释器组件，可以实现Java字节码和计算机操作系统之间的通信。



Java 是跨平台的语言，真正执行的不是二进制代码，而是字节码。

Java 是跨平台的，而JVM 不是跨平台的（JVM 是由 C 语言编写的）

Java 之所以能够做到跨平台，本质原因在于JVM不是跨平台的。

## 第一个java程序

### HelloWorld.java

**public** **class** HelloWorld

{

**public** **static** **void** main(String[] args)

{

System.*out*.println("Hello world!");

}

}

程序解释：

* 关键字class用来定义一个类，Hello是类的名称；
* 在Java程序中，所有的语句都必须放在某个类中；
* 整个类及其所有成员都包含在一对花括号中（即{和}之间），它们标志着类定义块的开始和结束。
* 定义类的一般语法：

class 类名

{

……

}

* 程序必须从main函数开始执行；
* 关键字public是访问修饰符，用来控制类成员的可见范围及作用域；
* 关键字static允许在不创建类的实例的情况下，调用main函数；
* 关键字void用来说明main函数是没有返回值的；
* args是字符串数组，用来从命令行接收参数；
* main函数的一般形式如下：

**public** **static** **void** main(String[] args)

{

……

}

* String[]是字符串类型的数组；
* 在Java中，类名称一般首字母大写；
* 使用System.out.println()将信息输出到控制台上，即打印出来。

### java程序执行步骤

1、编译。从命令行进入代码目录，执行如下命令：

javac HelloWorld.java

2、执行。Class文件是字节码文件，程序最终执行的就是这个字节码（ bytecode）文件。

执行命令： java HelloWorld（注意， Test 后面没有.class）

## java数据类型

### Java中的数据类型分为两大类：

1）原生数据类型（Primitive Data Type）

2）引用类型（对象类型）（Reference Type）

Java编程语言定义了八种基本的数据类型，共分为四类：整数类(byte、short、int、long)、文本类(char)、浮点类(double、float)和逻辑类(boolean)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 分 类 | 数据类型 | 关键字 | 占字节数 | 缺省数值 | 取 值 范 围 |
| 整数类 | 字节型 | byte | 8 | 0 | -2 7～2 7-1 |
| 短整型 | short | 16 | 0 | -2 15～2 15-1 |
| 整型 | int | 32 | 0 | -2 31～2 31-1 |
| 长整型 | long | 64 | 0 | -2 63～2 63-1 |
| 文本类 | 字符型 | char | 16 | ′\u 0000′ | ′\u 0000′ ～ ′\u FFFF′ |
| 浮点类 | 浮点型 | float | 32 | 0.0F | — |
| 双精度型 | double | 64 | 0.0D | — |
| 逻辑类 | 逻辑型 | boolean | 8 | False | True、False |

### 强制类型转换

Java 中的所有浮点类型默认情况下都是double。 不能将double 类型的值赋给float类型的变量，即便该 double 类型的值处于float 类型的范围内也是不可以的。能否成功赋值取决于等号右边的值类型与等号左边的变量类型是否一致。

如何将 double 类型的值赋给 float 类型的变量？

(1)强制类型转换，将 double 类型的值强制转换为 float 类型。

(2)使用 java 语言的支持。

**public** **class** Variable {

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

**double** a = 1.2;

System.***out***.println(a);

// 这行代码编译出错:Type mismatch: cannot convert from double to float

// float b = 1.2;

// System.out.println(b);

//1)强制类型转换

**float** c = (**float**) 1.2;

System.***out***.println(c);

//2)使用 java语言的支持，大小写F均可以

**float** d = 1.2F;

System.***out***.println(d);

}

}

强制转换的语法： 类型 变量名 = （类型） 变量值;

### 变量赋值规则

如下代码可以通过编译：

short a = 1;

int b = a;

a 是short 类型， b是 int 类型， int类型表示的数据范围要比short 类型大，可以将表示范围小的值赋给表示范围大的变量。

总结：可以将表示范围小的值赋给表示范围大的变量；但不能直接将表示范围大的值赋给表示范围小的变量，只能通过强制类型转换实现。

## 运算符

### 运算符取值原则

当有若干个变量参与运算时， 结果类型取决于这些变量中表示范围最大的那个变量类型。比如，参与运算的变量中，有整型 int，有双精度浮点型 double，有短整型 short，那么最后的结果类型就是 double

**public** **class** Calc {

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

**int** a = 1;

**int** b = 2;

**int** c = a / b;

//此处打印0，小数点后的内容被舍弃了

System.***out***.println(c);

**int** d = 1;

**int** e = 2;

//此处要使用强制类型转换，否则会打印0.0

**float** f = (**float**)d / e;

//此处打印0.5

System.***out***.println(f);

}

}

a 与 b 都是整型，但是通过(double)a 这种转换将 a 转换为一个匿名的变量，该变量的类型是 double，但是要注意： a 本身依旧是 int 类型，而不是 double 类型，这样， (double)a / b 就是 double 类型除以 int 类型，结果自然是 double 类型。

### 取模运算符：使用 %表示。

int a = 5;

int b = 3;

int c = a % b;

上面代码的运行结果是2，因为5除以3结果是 1 余 2。

取模的规律：取模的结果符号永远与被除数的符号相同

int a = 5;

int b = -3;

int c = a % b;

被除数是5，那么取模的结果是 2

int a = -5;

int b = 3;

int c = a % b;

被除数是-5，那么取模的结果是-2

### 关系运算符

大于（>）、小于（<）、等于（==）、不等于（!=）、大于等于（>=）、小于等于（<=），关系运算的结果是个boolean值。

### 逻辑运算符

重点讲解两个，逻辑运算符本身也返回一个boolean 值。

1）逻辑与： 使用&&表示，逻辑与是个双目运算符（即有两个操作数的运算符），只有当两个操作数都为真的时候，结果才为真；其余情况结果均为假。逻辑与表示的并且的意思。

2）逻辑或：使用||表示，逻辑或也是个双目运算符，有当两个操作数都为假的时候，结果才为假；其余情况结果均为真。逻辑或表示或者的意思。

**逻辑运算符的短路特性**

1） 逻辑与：如果第一个操作数为false，那么结果肯定就是false，所以在这种情况下，将不会执行逻辑与后面的运算了，即发生了短路。

2） 逻辑或：如果第一个操作数为true，那么结果肯定就是true，所在在这种情况下，将不会执行逻辑或后面的运算了，即发生了短路。

**public** **class** LogicTest {

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

**int** a = 1;

**int** b = 2;

**int** c = 3;

**int** d = 4;

**int** e = 5;

// 逻辑与测试，可更改括号内的值进行测试

**boolean** f = (a > b) && (c < d);

System.***out***.println("f = " + f);

// 逻辑或测试，可更改括号内的值进行测试

**boolean** g = (a > b) || (c > d);

System.***out***.println("g = " + g);

// 逻辑与短路测试

**boolean** h = (a < b) && ((e = a) < b);

System.***out***.println("e = " + e);

System.***out***.println("h = " + h);

// 逻辑或短路测试

**boolean** i = (a < b) || ((e = a) < b);

System.***out***.println("e = " + e);

System.***out***.println("i = " + i);

}

}

### 条件运算符

三目运算符，语法格式如下：

type d = (expression) ? if-true-statement : if-false-statement;

三目运算符的规则是：先对逻辑表达式expression取值，如果expression返回true，则返回第二个操作数的值，如果expression返回false，则返回第三个操作数的值。;

### 变量的自增与自减运算

1） int b = a++，作用是将a的值先赋给b，然后再让a自增1.

2） int b = ++a，作用是将a的值先自增1，然后将自增后的结果赋给b。

**public** **class** SelfAdd {

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

**int** a = 3;

**int** b = a++;

System.***out***.println("a = " + a);

System.***out***.println("b = " + b);

**int** c = 3;

**int** d = ++c;

System.***out***.println("c = " + c);

System.***out***.println("d = " + d);

}

}

## 类与对象

* **类**

人们为了更好地认识世界，将现实生活中的事物（对象）划分成类。同一类中的事物总是具有一些共性。类以共同的特性和行为定义实体，类是具有相同属性和和行为的一组对象的集合。

定义类的语法

class 类名

{

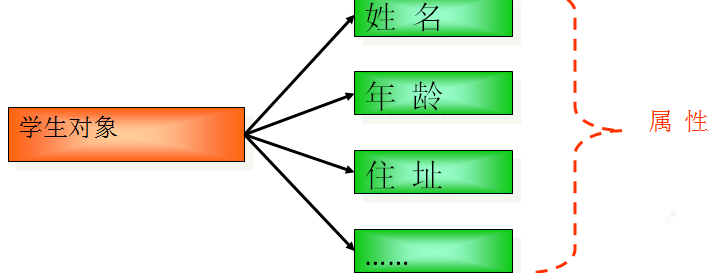
成员列表

//包括成员变量和成员函数

}

属性

事物的特性在类中用变量表示，每个对象的每个属性都拥有其特定的值，属性名称由类的所有对象共享，对象或实体拥有的特征在类中表示时称为属性。



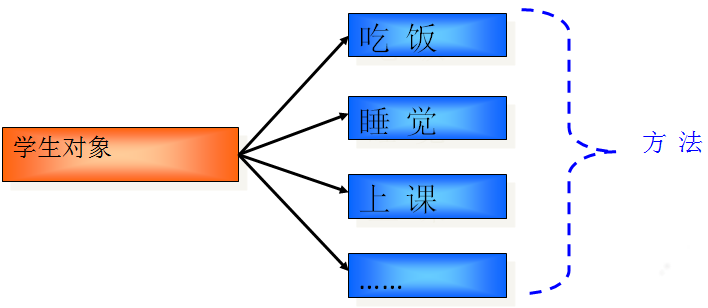
方法

事物的行为和动作在类中用函数表示，每个对象都有相同的动作和行为，对象执行的操作在类中表示为方法

main方法是整个java程序的入口点，如果类的定义中没有main方法，则程序无法执行

方法定义不能嵌套，不能再一个方法中定义另外一个方法，方法只能定义在类中

方法的执行：首先定义方法，接下来就可以使用方法(调用方法)，当方法调用完毕后，方法可以返回值。方法到底是否有返回值是由方法的定义决定的。

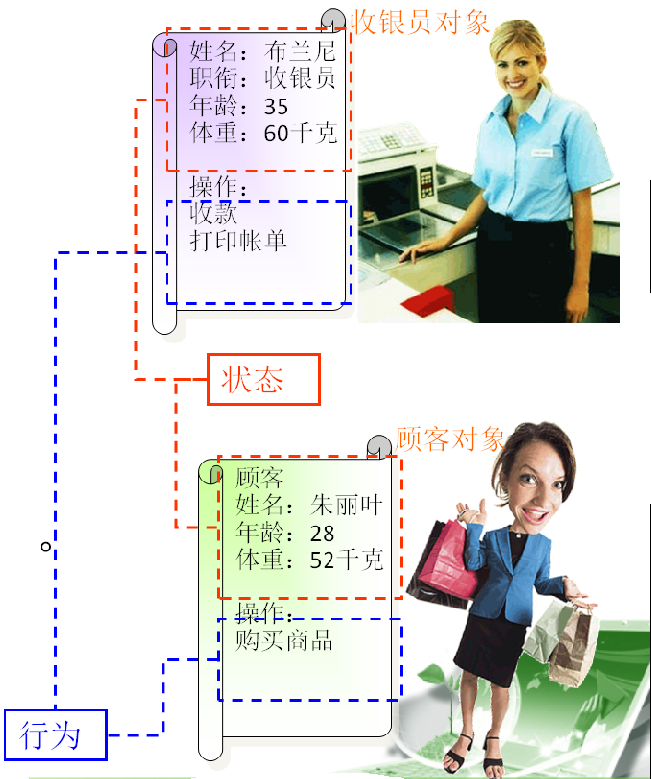


* **对象**

对象是面向对象编程的核心部分，是实际存在的具体实体，具有明确定义的状态和行为。对象其实就是“数据”和“函数”的封装体，其中：

数据表示自身的状态，也称作“属性”或“成员数据”；

函数表示自身的功能，也称作“方法”或“成员函数”。



类和对象的区别

1. 类是用来描述实体的“模板”或“原型”；
2. 对象是实际的实体，每一个对象都是类的一个具体实例；
3. 类用来定义对象所有的属性和方法，同一类的所有对象都拥有相同的特征和操作；
4. 可以将类理解成生产产品的模具，而对象则是根据此模具生产的一个个产品。

# 封装

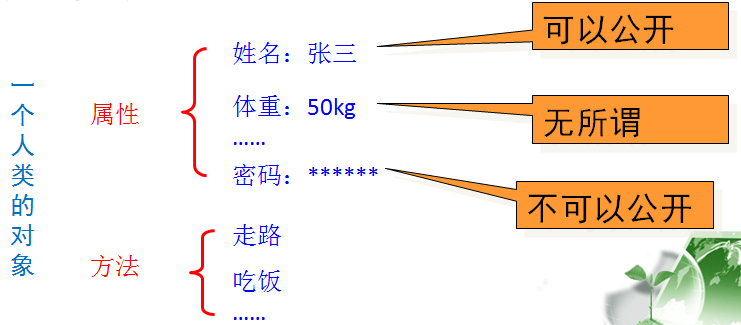
面向对象的程序设计有三大特征：

封装：解决了数据的安全性问题

继承：解决了代码的重用问题

多态：解决了程序的扩展问题

将某些东西包装在一起，然后以新的完整形式呈现出来，隐藏属性、方法或实现细节的处理方式称为封装，封装其实就是有选择性地公开或隐藏某些信息，它解决了数据的安全性问题。



代码示例：

**public** **class** EncapsulationTest

{

**public** **static** **void** main(String[] args)

{

Calculator cal = **new** Calculator();

cal.setNum1(1);

**int** a = cal.getNum1();

cal.setNum2(2);

**int** b = cal.getNum2();

System.*out*.println("a + b = " + cal.add(a, b));

}

}

**class** Calculator

{

**int** num1;

**int** num2;

// 定义num1的set和get方法

**public** **int** getNum1()

{

**return** num1;

}

**public** **void** setNum1(**int** num1)

{

**this**.num1 = num1;

}

// 定义num2的set和get方法

**public** **int** getNum2()

{

**return** num2;

}

**public** **void** setNum2(**int** num2)

{

**this**.num2 = num2;

}

**public** **int** add(**int** num1, **int** num2)

{

**return** num1 + num2;

}

}

## 构造方法

构造方法用于完成对象的初始化工作。没有构造方法时，只能逐个地为数据成员赋值，如果想在对象实例化的同时就初始化成员属性，就使用到了构造方法；构造方法是特殊的成员方法，它与类同名，在对象实例化时由虚拟机自动调用；构造方法的特点：

1. 构造方法的名字必须与类名完全一致（包括大小写）
2. 构造方法没有返回值，也不能有返回值，连void也不能出现
3. 如果在定义一个类的时候，没有为类声明构造方法，那么java编译器会自动为类添加一个没有参数且方法体为空的构造方法（默认的构造方法）；如果在定义一个类的时候，为类声明了构造方法，那么java编译器就不会再为类添加构造方法了
4. 默认的构造方法：构造方法没有参数且方法体为空
5. 不能显式调用类的构造方法，构造方法通常是通过new关键字隐式调用
6. 每个对象在生成时都必须执行构造方法，而且只能执行一次；如果构造方法调用失败，那么对象也无法创建

没有使用构造方法的学生类，Test.java

**public** **class** Test

{

**public** **static** **void** main(String[] args)

{

// 实例化学生类对象：张三

Student std1 = **new** Student();

std1.setName("张三");

std1.setAge(18);

std1.setWeight(50);

std1.display();

//实例化学生类对象：李四

Student std2 = **new** Student();

std2.setName("李四");

std2.setAge(20);

std2.setWeight(55);

std2.display();

}

}

//定义学生类

**class** Student

{

//学生的几个属性，私有

**private** String name;

**private** **int** age;

**private** **float** weight;

// 为学生的姓名赋值的方法，公有

**public** **void** setName(String name)

{

**this**.name = name;

}

// 为学生的年龄赋值的方法，公有

**public** **void** setAge(**int** a)

{

age = a;

}

// 为学生的体重赋值的方法，公有

**public** **void** setWeight(**float** w)

{

weight = w;

}

// 将学生的所有信息打印出来的方法，公有

**public** **void** display()

{

System.*out*.println("姓名：" + name + ", 年龄：" + age + ", 体重：" + weight);

}

}

分析Test.java的代码可见，没有构造时，逐个为对象赋值是一件比较麻烦的事，修改代码使用构造方法：

**public** **class** Test

{

**public** **static** **void** main(String[] args)

{

// 实例化学生类对象：张三

Student std1 = **new** Student("张三",18,50);

std1.display();

//实例化学生类对象：李四

Student std2 = **new** Student("李四",20,55);

std2.display();

}

}

//定义学生类

**class** Student

{

//学生的几个属性，私有

**private** String name;

**private** **int** age;

**private** **float** weight;

//定义学生类的构造方法

**public** Student(String name,**int** age,**float** weight)

{

**this**.name = name;

**this**.age = age;

**this**.weight = weight;

}

// 将学生的所有信息打印出来的方法，公有

**public** **void** display()

{

System.*out*.println("姓名：" + name + ", 年龄：" + age + ", 体重：" + weight);

}

}

## 方法重载(Overload)

方法重载用于定义一组方法，这些方法具有**相同的名称**，并执行类似的操作，但是却使用不**同的参数列表**；编译器会根据调用时**传递的实际参数**自动判断具体调用的是哪个重载方法。

### 方法重载的三大原则

1. 方法名相同
2. 参数不同，可以有三方面的不同：数量不同、类型不同、顺序不同
3. 同一作用域

代码示例：

AbsoluteValue.java 求出一个整数和一个浮点数的绝对值：

**public** **class** AbsoluteValue

{

/\*求整数绝对值的方法\*/

**public** **int** absOfInt(**int** x)

{

**return** ((x >= 0) ? x : -x);

}

/\*求浮点数绝对值的方法\*/

**public** **float** absOfFloat(**float** x)

{

**return** ((x >= 0) ? x : -x);

}

**public** **static** **void** main(String[] args)

{

AbsoluteValue obj = **new** AbsoluteValue();

**int** a = -8, b;

b = obj.absOfInt(a);

System.*out*.println(a + "的绝对值是" + b);

**float** c = -3.14f, d;

d = obj.absOfFloat(c);

System.*out*.println(c + "的绝对值是" + d);

}

}

由以上代码可见，如果还要求其它类型数据的绝对值，就需要定义更多的方法，这对于调用者来说，将会是非常麻烦的事情。可以使用构造方法对代码进行改进：

**public** **class** AbsoluteValue

{

/\*求整数绝对值的方法\*/

**public** **int** abs(**int** x)

{

**return** ((x >= 0) ? x : -x);

}

/\*求浮点数绝对值的方法\*/

**public** **float** abs(**float** x)

{

**return** ((x >= 0) ? x : -x);

}

**public** **static** **void** main(String[] args)

{

AbsoluteValue obj = **new** AbsoluteValue();

**int** a = -8, b;

b = obj.abs(a);

System.*out*.println(a + " 的绝对值是 " + b);

**float** c = -3.14f, d;

d = obj.abs(c);

System.*out*.println(c + " 的绝对值是 " + d);

}

}

### 方法重载与返回值

方法重载跟方法的返回值类型没有任何关系。也就是说，只有返回值不同的方法不能构成重载。

**class** AbsoluteValue

{

/\*求整数绝对值的方法\*/

**public** **int** abs(**int** x)

{

**return** ((x >= 0) ? x : -x);

}

/\*打印整数绝对值的方法\*/

**public** **void** abs(**int** x)

{

System.*out*.println(x);

}

}

上面的程序将会编译错误：



### 重载构造方法

方法重载的一个重要应用就是重载构造方法；可以定义一组构造方法，使其构成重载关系，从而能够灵活的以各种不同的方式来实例化对象。

**public** **class** People

{

**public** **static** **void** main(String[] args)

{

Play paly1 = **new** Play("小明");

Play paly2 = **new** Play("小明","小军");

Play paly3 = **new** Play("小明","小军","小强");

}

}

**class** Play

{

**public** Play(String name)

{

System.*out*.println(name+"可以玩三国志11");

}

**public** Play(String name,String name1)

{

System.*out*.println(name+"和"+name1+"可以玩象棋");

}

**public** Play(String name,String name1,String name2)

{

System.*out*.println(name+"、"+name1+"和"+name2+"可以玩斗地主");

}

}

如果想在一个构造方法中，调用另外一个构造方法，可以使用this关键字

**public** **class** OverloadTest

{

**public** **static** **void** main(String[] args)

{

Constructor co = **new** Constructor();

}

}

**class** Constructor

{

**public** Constructor()

{

**this**(1);

System.*out*.println("这是不带参数的构造方法");

}

**public** Constructor(**int** i)

{

System.*out*.println("i = " + (++i) + " : 这是传入整形参数的构造方法");

}

}

上面代码的输出如下：

i = 2 : 这是传入整形参数的构造方法

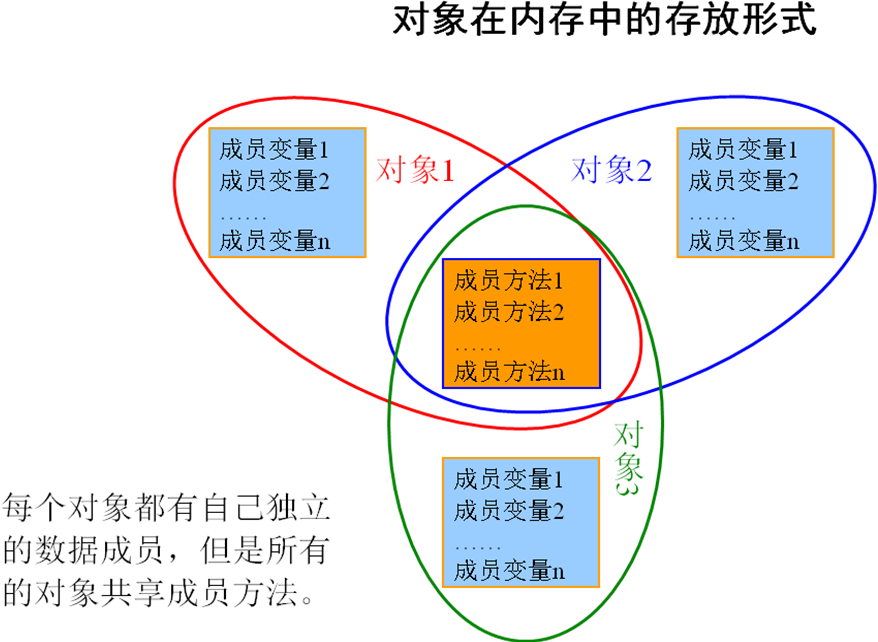
这是不带参数的构造方法

## this关键字

this 属于一个具体的对象，而不是属于类。相当于我的。如 this.name 我的名字

如下图所示，每个对象都有自己独立的成员变量，但是所有的对象共享成员方法。既然所有的对象都共用相同的成员方法，那么在不同的对象都调用同一方法时，它怎么确定要使用哪个对象的数据成员呢？

1. 每个成员方法都有一个隐含的this引用，它总是指向调用它的对象；
2. 关键字this给出用于调用成员方法的对象的地址；
3. 每当调用成员方法时，编译器会向this分配调用该方法的对象的地址；
4. 可以像使用任何引用那样使用this。



代码示例：

**public** **class** Student

{

**private** String mName;

**private** **int** age;

**public** Student(String name, **int** age)

{

//隐式调用，等同于this.mName = name;

mName = name;

//显式调用，等同于mAge = age;

**this**.age = age;

}

}

**class** Person

{

String name;

**int** age;

Dog dog;

**public** Person(Dog dog,String name,**int** age)

{

**this**.name = name;

**this**.age = age;

**this**.dog = dog;

}

**public** **void** showinfo()

{

System.*out*.println("人的名字是： "+**this**.age);

}

}

**class** Dog

{

String name;

**int** age;

**public** Dog(String name,**int** age)

{

**this**.name = name;

**this**.age = age;

}

**public** **void** showinfo()

{

System.*out*.println("狗的名字是： "+**this**.age);

}

}

# 继承

继承是利用现有的类创建新类的过程，现有的类称作基类（或父类），创建的新类称作派生类（子类）。最高层是最普遍的、最一般的情况，往下每一层都比上一层更具体，并包含有高层的特征，通过这样的层次结构使下层的类能自动享用上层类的特点和性质；继承其实就是自动地共享基类中成员属性和成员方法的机制。

继承的作用：

当今软件设计的特征：

软件规模越来越大；

软件设计者越来越多；

软件设计分工越来越细。

引入继承，实现了代码重用；

引入继承，实现了递增式的程序设计。

继承是能自动传播代码和重用代码的有力工具；

继承能够在某些比较一般的类的基础上建造、建立和扩充新类；

能减少代码和数据的重复冗余度，并通过增强一致性来减少模块间的接口和界面，从而增强了程序的可维护性；

能清晰地体现出类与类之间的层次结构关系。

注意事项

继承是单方向的，即派生类可以继承和访问基类中的成员，但基类则无法访问派生类中的成员；

在Java中只允许单一继承方式，即一个派生类只能继承于一个基类，而不能象C++中派生类继承于多个基类的多重继承方式。

代码示例，InheritanceTest.java

派生类Apple自动继承了基类Fruit的成员属性(name)和成员方法(showinfo())

**public** **class** InheritanceTest

{

**public** **static** **void** main(String[] args)

{

Apple apple = **new** Apple();

System.*out*.println(apple.name);

apple.showinfo();

}

}

**class** Fruit

{

String name = "Fruit";

**public** **void** showinfo()

{

System.*out*.println("水果熟了");

}

}

**class** Apple **extends** Fruit

{

}

## 访问控制权限

类成员的访问控制权限

信息隐藏是面向对象程序设计的重要特点之一，它可以：防止类的使用者意外损坏数据；对任何实现细节所作的修改不会影响到使用该类的其它代码；使类更易于使用。

在Java中实现信息隐藏的是访问控制权限机制；

访问控制权限包括4个访问修饰符：public、protected、private和缺省；可以使用上述访问修饰符修饰类的成员。

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

详细的访问控制权限列表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 访问修饰符  位置 | public | protected | 缺省 | private |
| 本类 | 可以 | 可以 | 可以 | 可以 |
| 同包子类 | 可以 | 可以 | 可以 | 不可以 |
| 同包非子类 | 可以 | 可以 | 可以 | 不可以 |
| 不同包子类 | 可以 | 可以 | 不可以 | 不可以 |
| 不同包非子类 | 可以 | 不可以 | 不可以 | 不可以 |

类的访问权限

可以在定义类时为类添加访问修饰符，对类进行访问权限控制：

1. 对类使用的访问修饰符只有public和缺省两种；
2. 被public修饰的类可以从任何地方访问，不受限制；
3. 不加访问修饰符，缺省修饰的类只能从本包中访问，不同包则无法访问到；

注意：在一个源文件中只能有一个被public修饰的类，并且文件名必须与public的类同名；如果要定义多个public的类，则必须分别写在不同的源文件中，**一个源文件只写一个类是良好的编程习惯**

## 继承中的构造方法

当实例化子类的对象时，必须先执行父类的构造方法，然后再执行子类的构造方法；如果父类还有更上级的父类，就会先调用最高父类的构造方法，再逐个依次地将所有继承关系的父类构造方法全部执行。

父类中的构造方法不能被子类继承，即便它是public的；父类的构造方法负责初始化属于它的成员变量，而子类的构造方法则只需考虑属于自己的成员变量，不必去关注父类的情况。

1. 当生成子类对象时，Java默认首先调用**父类的不带参数**的构造方法，生成父类的对象。接着再调用子类的构造方法，生成子类的对象。

**class** Fruit

{

// public Fruit()

// {

// System.out.println("这是父类不带参数的构造方法");

// }

**public** Fruit(**int** i)

{

System.*out*.println("这是父类接收int i 的构造方法");

}

}

**class** Apple **extends** Fruit

{

**public** Apple()

{

//super(1);

System.*out*.println("这是子类不带参数的构造方法");

}

}

代码将会编译错误：



1. 生成子类的对象时，父类的构造方法也会被调用

代码示例：InheritanceTest.java 在创建son对象时，类Grandpa和Father的构造方法都得到了执行

**public** **class** InheritanceTest

{

**public** **static** **void** main(String[] args)

{

Son son = **new** Son();

}

}

**class** Grandpa

{

**public** Grandpa()

{

System.*out*.println("this is grandpa's constructor");

}

}

**class** Father **extends** Grandpa

{

**public** Father()

{

System.*out*.println("this is father's constructor");

}

}

**class** Son **extends** Father

{

**public** Son()

{

System.*out*.println("this is son's constructor");

}

}

1. 如果父类的构造方法执行失败，那么子类的对象也将无法实例化。如下代码，必须在子类的构造方法中，使用super关键字显式地调用父类的构造方法，将参数传递给它；

**super关键字必须是构造方法的第一行语句。**

**public** **class** Demo

{

**public** **static** **void** main(String[] args)

{

Circle c = **new** Circle(2.5f);

}

}

**class** Point

{

// x轴坐标和y轴坐标，由于准备用于继承，故修饰为protected

**protected** **float** x , y;

**public** Point(**float** x, **float** y)

{

**this**.x = x;

**this**.y = y;

}

}

**class** Circle **extends** Point

{

**protected** **float** radius;

**public** Circle(**float** r)

{

**super**(1.2f,1.2f); //无此行的话，程序将不会编译通过，super(1.2 , 1.2)也不行

**this**.radius = r;

}

}

1. 子类的构造方法隐式的调用父类的不带参数的构造方法，当父类没有不带参数的构造方法时，子类需要使用super来显示的调用父类的构造方法，super指的是对父类的引用。

## super关键字

作用一：在子类的构造方法中，super关键字可以显式地调用父类的构造方法，用于将参数传递给它。其一般语法是：super(实际参数)；

注意：该语句必须是子类构造方法的第一条语句。

作用二：如果父类和子类中有同名成员，在子类中默认访问是属于自己的那一个成员；super关键字可以明确地指定要访问父类中的成员；

其一般语法是： super.成员名;

前提条件是：父类中的该成员不是private的。

但是，super并不表示一个指向对象的引用，它只是一个特殊的关键字，用来告诉编译器，现在要调用的是父类的方法。

super与this的比较

this关键字有两种意义：

表示一个指向“implicit parameter”的引用

调用本类中的另一个构造器

super关键字也有两种意义：

调用父类的方法

调用父类的构造器

## static关键字

1. 静态成员变量

在成员变量前加static关键字，可以将其声明为静态成员变量，可以通过类名.成员变量名的方式访问静态成员变量；

1. 如果类中成员变量被定义为静态，那么不论有多少个对象，静态成员变量只有一份内存拷贝，即所有对象共享该成员变量；
2. 静态成员变量的作用域只在类内部，但其生命周期却贯穿整个程序；
3. 在没有实例化对象时，可以通过类名访问静态成员变量；也可以通过对象访问静态成员变量，但不论使用的是哪个对象，访问到的都是同一个变量；

**public** **class** StaticTest

{

**public** **static** **void** main(String[] args)

{

MyStatic.*a* = 10;

System.*out*.println(MyStatic.*a*);

}

}

**class** MyStatic

{

**static** **int** *a*;

}

1. 静态成员变量在声明时最好初始化，如果不进行初始化，系统会默认初始化为初始值。

代码示例： StaticTest.java 所有狗对象共享静态成员变量count

**public** **class** StaticTest

{

**public** **static** **void** main(String[] args)

{

System.*out*.println("当前狗的数量是：" + Dog.*count*); **//打印值为0**

Dog d1 = **new** Dog();

Dog d2 = **new** Dog();

System.*out*.println("当前狗的数量是：" + Dog.*count*); **//打印值为2**

}

}

**class** Dog

{

// 静态成员变量

**public** **static** **int** *count* = 0;

// 构造方法

**public** Dog()

{

*count*++;

}

}

不论哪个对象，访问到的都是同一个变量，代码示例：

不含static关键字的代码：

**public** **class** TestStatic

{

**public** **static** **void** main(String[] args)

{

Dog dog1 = **new** Dog();

Dog dog2 = **new** Dog();

dog1.age = 2;

System.*out*.println(dog2.age); // 打印值为0

}

}

**class** Dog

{

**int** age;

}

含有static关键字的代码：

**public** **class** TestStatic

{

**public** **static** **void** main(String[] args)

{

Dog dog1 = **new** Dog();

Dog dog2 = **new** Dog();

dog1.*age* = 2;

System.*out*.println(dog2.*age*); //打印值为2

System.*out*.println(Dog.*age*); //**静态成员变量推荐使用这种方法进行访问**

}

}

**class** Dog

{

**static** **int** *age*;

}

1. 静态成员方法

在成员方法前加static关键字，可以将其声明为静态成员方法。可以通过类名.方法名的方式来访问静态成员方法

1. 在没有实例化对象时，可以通过类名访问静态成员方法

**public** **class** StaticTest

{

**public** **static** **void** main(String[] args)

{

MyStatic.*output*(); **// 程序输出this is a static method output**

}

}

**class** MyStatic

{

**public** **static** **void** output()

{

System.*out*.println("this is a static method output");

}

}

1. 静态方法只能继承，不能重写（override）。子类不能重写父类的静态方法，也无法使用静态方法重写父类的方法。如下代码所示，子类隐藏了父类的testClassMethod方法，而不是重写。任意去掉父类或子类testClassMethod方法的static关键字，程序都会编译错误

**public** **class** StaticTest

{

**public** **static** **void** main(String[] args)

{

Animal animal = **new** Cat();

animal.*testClassMethod*(); **//调用的是父类的testClassMethod方法**

animal.testInstanceMethod(); **//调用的是子类的testInstanceMethod方法**

}

}

**class** Animal

{

**public** **static** **void** testClassMethod()

{

System.*out*.println("the class method in animal;");

}

**public** **void** testInstanceMethod()

{

System.*out*.println("the instance method in animal;");

}

}

**class** Cat **extends** Animal

{

**public** **static** **void** testClassMethod()

{

System.*out*.println("the class method in cat;");

}

**public** **void** testInstanceMethod()

{

System.*out*.println("the instance method in cat;");

}

}

1. 静态成员方法只能对类的静态成员变量进行操作，不能在静态方法中访问非静态成员变量（无法知道静态方法操作的是哪一个对象的成员变量）。同样，静态成员方法没有this引用。

代码示例，程序编译时将会报错：：

**class** Dog

{

**int** age = 10;

**public** **static** **void** change()

{

age++;



}

}

以下程序编译时也会报错：

**class** Dog

{

**int** age = 10;

**public** **static** **void** change()

{

**int** b = **this**.age;



}

}

**总结：静态方法只能访问静态的成员变量，非静态方法都可以访问。**

1. 静态代码块

静态代码块也是完成一些初始化工作。

1. 首先执行静态代码块，然后执行构造方法。静态代码块在类被加载时执行，而构造方法是在生成对象时执行。要想调用某个类来生成对象，首先需要将类加载到Java虚拟机上(JVM)，然后由JVM加载这个类来生成对象。
2. 类的静态代码块只会执行一次，是在类被加载的时候执行的。因为每个类只会被加载一次，所以静态代码块也只会被执行一次；而构造方法则不然，每次生成一个对象的时候都会调用类的构造方法，所以new一次就会调用构造方法一次。
3. 如果继承体系中既有构造方法，又有静态代码块；那么首先执行最顶层的类的静态代码块，一直执行到最底层的静态代码块，然后再去执行最顶层的类的构造方法，一直执行到最底层的类的构造方法。注意：生成多个对象时，静态代码块只会执行一次，构造方法会被多次调用。

代码示例：

**public** **class** StaticTest

{

**public** **static** **void** main(String[] args)

{

**new** S();

**new** S();

}

}

**class** P

{

**static**

{

System.*out*.println("P static block");

}

**public** P()

{

System.*out*.println("P constructor");

}

}

**class** Q **extends** P

{

**static**

{

System.*out*.println("Q static block");

}

**public** Q()

{

System.*out*.println("Q constructor");

}

}

**class** S **extends** Q

{

**static**

{

System.*out*.println("S static block");

}

**public** S()

{

System.*out*.println("S constructor");

}

}

代码执行结果为：

P static block

Q static block

S static block

P constructor

Q constructor

S constructor

P constructor

Q constructor

S constructor

# 多态

多态：父类型的引用可以指向子类型的对象

## 方法覆盖

在类的继承体系结构中，如果子类中出现了与父类中有同原型的方法，那么认为子类中的方法覆盖了父类中的方法（也称为方法重写）；

通过子类的实例调用被覆盖的方法时，将总是调用子类中的方法，而父类中的方法将被隐藏。

代码示例：

**public** **class** OverriddenTest

{

**public** **static** **void** main(String[] args)

{

Parent parObj = **new** Parent();

parObj.fun();

Child chiObj = **new** Child();

chiObj.fun();

}

}

**class** Parent

{

**public** **void** fun()

{

System.*out*.println("这是父类中的方法。");

}

}

**class** Child **extends** Parent

{

//子类覆盖父类中的fun方法

**public** **void** fun()

{

System.*out*.println("这是子类中的方法。");

}

}

方法覆盖的注意事项

子类中重写的方法，其访问权限不能比父类中被重写方法的访问权限更低

在子类中重写方法时要保持方法的签名与父类中方法的签名一致

## 方法重载与方法重写的区别

方法覆盖（重写）和方法重载是两个极易混淆的概念，必须严格区分；

方法覆盖出现的前提条件之一是必须有继承发生的情况下，而且要求父类和子类中的方法必须同原型；

方法重载时，继承并不是必需的，它只要求方法名称相同，而参数列表则必须不同，换言之，各方法的原型其实是不同的。

子类重载了父类的方法

**class** BaseClass // 定义基类

{

**public** **void** fun()

{

}

}

**class** DerivedClass **extends** BaseClass // 派生类继承于基类

{

**public** **void** fun(**int** x) // 跟父类中有相同名称的方法

{

}

}

子类重写了父类的方法

**class** SuperClass

{

**public** **void** fun()

{

}

}

**class** SubClass **extends** SuperClass

{

**public** **void** fun()

{

}

}

多态示例：

**public** **class** PolymorphismTest

{

**public** **static** **void** main(String[] args)

{

Shapes obj = **new** Shapes(); // 父类的引用指向父类的实例

obj.draw();

obj = **new** Circle(); // 父类的引用指向子类的实例

obj.draw();

obj = **new** Square(); // 父类的引用指向子类的实例

obj.draw();

}

}

**class** Shapes

{

**public** **void** draw()

{

System.*out*.println("绘制了一个基本形状。");

}

}

**class** Circle **extends** Shapes

{

**public** **void** draw()

{

System.*out*.println("绘制了一个圆形。");

}

}

**class** Square **extends** Shapes

{

**public** **void** draw()

{

System.*out*.println("绘制了一个正方形。");

}

}

从上例中可以看出，父类的引用指向哪个类的实例就调用哪个类中的方法；

同样是使用父类的引用，调用同一个名称的方法，却可以得到不同的调用结果，这就是Java中的多态，即：同一函数，多种形态；

静态多态

静态多态也称为编译时多态，即在编译时决定调用哪个方法。静态多态一般是指方法重载；只要构成了方法重载，就可以认为形成了静态多态的条件；静态多态与是否发生继承没有必然联系。

动态多态

动态多态也称为运行时多态，即在运行时才能确定调用哪个方法。形成动态多态必须具体以下条件：

1. 必须要有继承的情况存在；
2. 在继承中必须要有方法覆盖；
3. 必须由基类的引用指向派生类的实例，并且通过基类的引用调用被覆盖的方法；

由上述条件可以看出，继承是实现动态多态的首要前提。

## final关键字

final关键字可以用来修饰属性、方法、类

### 修饰属性

**该属性不能被改写**

当final修饰一个原生数据类型时，表示该原生数据类型的值不能发生变化，代码示例：FinalTest.java 代码中age的值不能再被改变

**public** **class** FinalTest

{

**public** **static** **void** main(String[] args)

{

People people = **new** People();

System.*out*.println(people.age);

people.age = 30; //the final field people.age cannot be assigned

}

}

**class** People

{

**final** **int** age = 20;

}

当final修饰一个引用类型时，表示该引用类型不能再指向其它对象了，但该引用所指向的内容是可以变化的。

代码示例：address再指向其它对象时，程序会编译错误

**public** **class** FinalTest

{

**public** **static** **void** main(String[] args)

{

People people = **new** People();

people.address = **new** Address(); //the final field people.address cannot be assigned

}

}

**class** People

{

**final** Address address = **new** Address();

}

**class** Address

{

String name = "beijing";

}

但是people.address 引用指向的内容是可以改变的:

**public** **class** FinalTest

{

**public** **static** **void** main(String[] args)

{

People people = **new** People();

people.address.name = "beijing";

System.*out*.println(people.address.name); //打印内容为北京

}

}

**class** People

{

**final** Address address = **new** Address();

}

**class** Address

{

String name = "beijing";

}

### 修饰方法

**该方法是一个终态方法，不能被重写（override）**

**public** **class** FinalTest

{

**public** **static** **void** main(String[] args)

{

Son son = **new** Son();

son.output();

}

}

**class** Father

{

**public** **final** **void** output()

{

System.*out*.println("Father");

}

}

**class** Son **extends** Father

{

**public** **void** output()



{

System.*out*.println("Son");

}

}

### 修饰类

**该类是一个终态类，不能被继承**

**public** **class** FinalTest

{

**public** **static** **void** main(String[] args)

{

}

}

**final** **class** Father // 该类不能被继承

{

}

**class** Son **extends** Father



{

}

### 常量声明

常量在声明必须初始化，一般来说有两种赋初值方式：

* 在声明final类型的成员变量时就赋上初值
* 在声明final类型的成员变量时不赋初值，但在类的**所有构造方法中**都为其赋上初值

**public** **class** FinalTest

{

**final** **int** a;

// 类FinalTest的第一个构造方法，为a赋初值

**public** FinalTest()

{

a = 0;

}

// 类FinalTest的第二个构造方法，也要为a赋初值

**public** FinalTest(**int** a)

{

**this**.a = a ;

}

**public** **static** **void** main(String[] args)

{

FinalTest ft1 = **new** FinalTest();

System.*out*.println(ft1.a);

FinalTest ft2 = **new** FinalTest(2);

System.*out*.println(ft2.a);

}

}

## 强制类型转换

### 向上类型转换（upcast）

**如下代码所示，可以将cat类型转换为animal类型，即可将子类型转换为父类型，而不需要显示指定**

**public** **class** PolyTest

{

**public** **static** **void** main(String[] args)

{

Cat cat = **new** Cat();

Animal animal = (Animal)cat; **//(Animal)一般省略**

animal.sing(); **//打印结果为： "cat is singing"**

}

}

**class** Animal

{

**public** **void** sing()

{

System.*out*.println("animal is singing");

}

}

**class** Cat **extends** Animal

{

**public** **void** sing()

{

System.*out*.println("cat is singing");

}

}

### 向下类型转化（downcast）

**如下代码所示，可以将animal类型转换为cat类型，即父类型转换为子类型，此时需要进行强制类型转换。程序能够正常运行，需要两个必要条件：**

1. **animal类型的引用指向了cat类型的对象；**
2. **使用强制类型转换将animal类型转换为cat类型**

**public** **class** PolyTest

{

**public** **static** **void** main(String[] args)

{

Animal animal = **new** Cat();

Cat cat =(Cat) animal;

cat.sing(); **//打印结果为 "cat is singing"**

}

}

**class** Animal

{

**public** **void** sing()

{

System.*out*.println("animal is singing");

}

}

**class** Cat **extends** Animal

{

**public** **void** sing()

{

System.*out*.println("cat is singing");

}

}

### 父类不能转换为子类的情况

**如下代码所示，animal型的引用指向animal类型的对象时，不能将animal类型的引用强制转换为cat类型的引用**

**public** **class** PolyTest

{

**public** **static** **void** main(String[] args)

{

Animal a = **new** Animal();

Cat b = (Cat)a;

b.sing();

}

}

**class** Animal

{

**public** **void** sing()

{

System.*out*.println("animal is singing");

}

}

**class** Cat **extends** Animal

{

**public** **void** sing()

{

System.*out*.println("cat is singing");

}

}

代码编译时不会出错，但是执行时会抛出异常：

Exception in thread "main" java.lang.ClassCastException: Animal cannot be cast to Cat at PolyTest.main(PolyTest.java:6)

如果不使用强制类型转换，则程序直接会报编译错误：

**public** **class** PolyTest

{

**public** **static** **void** main(String[] args)

{

Animal a = **new** Animal();

Cat b = a;



b.sing();

}

}

**class** Animal

{

**public** **void** sing()

{

System.*out*.println("animal is singing");

}

}

**class** Cat **extends** Animal

{

**public** **void** sing()

{

System.*out*.println("cat is singing");

}

}

### 不同子类间不能互相转化的情况

**如下代码所示，不能将类型为animal、指向dog类型的引用，强制转化为cat类型的引用**

**public** **class** PolyTest

{

**public** **static** **void** main(String[] args)

{

Animal a = **new** Dog();

Cat b = (Cat) a; **//这一行会抛出异常**

b.sing();

}

}

**class** Animal

{

**public** **void** sing()

{

System.*out*.println("animal is singing");

}

}

**class** Dog **extends** Animal

{

**public** **void** sing()

{

System.*out*.println("dog is singing");

}

}

**class** Cat **extends** Animal

{

**public** **void** sing()

{

System.*out*.println("cat is singing");

}

}

上述代码编译时不会出错，但是运行时会抛出异常：

Exception in thread "main" java.lang.ClassCastException: Dog cannot be cast to Cat at PolyTest.main(PolyTest.java:6)

### 使用多态调用方法时的强制类型转换

1. 当使用多态调用方法时，首先检查父类是否有sing()方法，如果没有则编译错误，如果有，再去调用子类的sing()方法

**public** **class** PolyTest

{

**public** **static** **void** main(String[] args)

{

Parent p = **new** Child();

p.sing();



}

}

**class** Parent

{

// public void sing()

// {

// System.out.println("parent is singing");

// }

}

**class** Child **extends** Parent

{

**public** **void** sing()

{

System.*out*.println("child is singing");

}

}

如上代码所示，当父类的sing()方法被注释后，程序将会编译错误。只有取消注释，程序才能正确运行，打印出 "child is singing"

1. 可以使用强制类型转换的方式，使用子类有而父类没有的sing()方法。如下代码所示，将p强制转换为child类型后，就可以使用child中的sing()方法了

**public** **class** PolyTest

{

**public** **static** **void** main(String[] args)

{

Parent p = **new** Child();

Child c = (Child)p;

c.sing();

}

}

**class** Parent

{

// public void sing()

// {

// System.out.println("parent is singing");

// }

}

**class** Child **extends** Parent

{

**public** **void** sing()

{

System.*out*.println("child is singing");

}

}

类与类之间的关系

类与类之间的关系一般来说有两种：“有”关系和“是”关系；

所谓“是”关系就是指类的继承关系。如：从动物类派生出哺乳动物类，那么可以说哺乳动物是动物；

而“有”关系是指在一个类中包含了另一个类的对象，即一个类中有另一个类（的对象），可以理解为类的嵌套。

## 抽象类、抽象方法

### 抽象类

使用了abstract关键字所修饰的类叫做抽象类，一般语法：

[访问权限] **abstract class** 类名

{

成员列表

}

如：

**public** **abstract** **class** Shapes

{

**public** **abstract** **void** draw();

}

### 抽象类不能实例化

如下代码所示，实例化抽象类Dog时，程序将会编译错误

**public** **class** AbstractTest

{

**public** **static** **void** main(String[] args)

{

Dog dog = **new** Dog();



}

}

**abstract** **class** Dog

{

}

抽象类只可以用来继承

**public** **class** AbstractTest

{

**public** **static** **void** main(String[] args)

{

Collie collie = **new** Collie();

collie.run(); //程序打印 collie is running

}

}

**abstract** **class** Dog

{

}

**class** Collie **extends** Dog

{

**public** **void** run()

{

System.*out*.println("collie is running");

}

}

### 抽象方法

在某些情况下，基类无法（或者没有必要）提供被覆盖方法的具体实现，那么就可以将此方法声明成抽象方法；使用关键字abstract声明抽象方法，一般语法：

[访问权限] abstract 返回值类型 方法名称(参数列表)；如：

**public**  **abstract** **void** draw();

注意是没有花括号的，即使空的花括号也不行。空的花括号表示有实现，只不过该实现是空的。

**abstract** **class** Dog

{

**public** **abstract** **void** run() {}



}

### 抽象方法需要定义在抽象类中

如下代码所示，将抽象方法run定义在了非抽象类dog中，程序会编译错误

**class** Dog



{

**public** **abstract** **void** run();



}

### 抽象类中也可以包含非抽象的方法

如下代码所示，抽象类dog中既有抽象方法run()，又有非抽象方法grow()

**abstract** **class** Dog

{

**public** **abstract** **void** run();

**public** **void** grow()

{

System.*out*.println("dog is growing");

}

}

### 抽象类的派生子类

抽象类的派生子类应该提供**对其所有抽象方法的具体实现**；如果抽象类的派生子类没有实现其中的**所有**抽象方法，那么该派生子类仍然是抽象类，**只能用于继承，而不能实例化**；

如下代码所示，当注释掉子类的run()方法后，程序会编译错误

**abstract** **class** Fruit

{

**public** **abstract** **void** run();

**public** **abstract** **void** sing();

**public** **void** grow()

{

System.*out*.println("dog is growing");

}

}

**class** Apple **extends** Fruit



{

// public void run()

// {

// System.out.println("Apple is runnig");

// }

**public** **void** sing()

{

}

}

也可以将apple定义为抽象类，再由它的子类实现抽象方法run()

**abstract** **class** Fruit

{

**public** **abstract** **void** run();

**public** **abstract** **void** sing();

**public** **void** grow()

{

System.*out*.println("dog is growing");

}

}

**abstract** **class** Apple **extends** Fruit

{

// public void run()

// {

// System.out.println("Apple is runnig");

// }

**public** **void** sing() { }

}

**class** BigApple **extends** Apple

{

**public** **void** run() { }

}

### 代码示例：抽象类的实际应用

**public** **class** AbstractTest

{

**public** **static** **void** main(String[] args)

{

Shape shape = **new** Triangle(10, 6);

**int** area = shape.computeArea();

System.*out*.println("triangle:" + area);

shape = **new** Rectangle(10, 10);

area = shape.computeArea();

System.*out*.println("rectangle:" + area);

}

}

**abstract** **class** Shape

{

**public** **abstract** **int** computeArea();// 计算形状面积

}

**class** Triangle **extends** Shape

{

**int** width;

**int** height;

**public** Triangle(**int** width, **int** height)

{

**this**.width = width;

**this**.height = height;

}

**public** **int** computeArea()

{

**return** (width \* height) / 2;

}

}

**class** Rectangle **extends** Shape

{

**int** width;

**int** height;

**public** Rectangle(**int** width, **int** height)

{

**this**.width = width;

**this**.height = height;

}

**public** **int** computeArea()

{

**return** **this**.width \* **this**.height;

}

}

可以这么认为，抽象方法实际上就是由抽象基类强制要求其派生子类必须实现的方法原型；

构造方法和静态方法不可以修饰为abstract。

## 接口

如果某个类中的所有方法都是抽象方法，那么可以考虑将该类定义为接口；

定义接口使用关键字interface，一般语法：

[访问权限] interface 接口名

{

成员列表

}

如：

**public** **interface** MyInterface

{

……

}

### 接口中的所有方法都必须是抽象方法

接口中不能定义非抽象方法，也就是说接口中不能包含有函数实体。如下代码所示，方法method3()含有函数主体，程序将会编译出错。

接口中的所有方法都默认为抽象方法，因此无需在每个方法前加abstract关键字。如下代码所示，方法method1()前并没有加abstract关键字。

**public** **interface** InterfaceTest

{

**public** **void** method1();

**public** **abstract** **void** method2();

**public** **void** method3() { }



}

### 接口无法实例化

与抽象类相似，接口同样不可以实例化，只能用于实现。如下代码所示，实例化接口Interface1时，程序会出现编译错误：

**public** **class** InterfaceTest

{

Interface1 in1 = **new** Interface1();



}

**interface** Interface1

{

**public** **void** method1();

**public** **abstract** **void** method2();

**public** **void** method3();

}

### 接口的实现

类可以**实现**接口。如果某类要实现接口，则使用implements关键字，一般语法：

[访问权限] **class** 类名 **implements** 接口名

{

成员列表

}

如：

**public** **class** MyClass **implements** IMyInterface

{

……

}

1. 接口的实现类应该提供对接口中所有抽象方法的具体实现，否则将成为抽象类；

如下代码所示，method3()没有被实现，程序会编译错误

**interface** Interface1

{

**public** **void** method1();

**public** **abstract** **void** method2();

**public** **void** method3();

}

**class** class1 **implements** Interface1

{

**public** **void** method1(){}

**public** **void** method2(){}

**public** **void** method3();



}

可以将class1声明为抽象类：

**interface** Interface1

{

**public** **void** method1();

**public** **abstract** **void** method2();

**public** **void** method3();

}

**abstract** **class** class1 **implements** Interface1

{

**public** **void** method1(){}

**public** **void** method2(){}

**public** **abstract** **void** method3();

}

1. Java只支持单继承，某个类只能继承于一个父类，但可以实现多个接口。代码示例：

**public** **class** InterfaceTest

{

**public** **static** **void** main(String[] args)

{

MyClass myClass = **new** MyClass();

myClass.output();

myClass.output2();

myClass.output3();

}

}

**interface** MyInterface

{

**public** **void** output();

}

**interface** MyInterface2

{

**public** **void** output2();

}

**class** MyParent

{

**public** **void** output3()

{

System.*out*.println("class MyParent output3");

}

}

**class** MyClass **extends** MyParent **implements** MyInterface, MyInterface2

{

**public** **void** output()

{

System.*out*.println("MyInterface output");

}

**public** **void** output2()

{

System.*out*.println("MyInterface2 output2");

}

}

### 接口的多态

与抽象类和它的继承类相似，也可以使用接口的引用指向其实现类的对象，从而达到多态的效果。接口与实现接口的类之间的强制类型转换方式，与父类和子类之间的强制转换方式完全一样。代码示例：

**public** **class** InterfaceTest

{

**public** **static** **void** main(String[] args)

{

/\*

BB bb = new BB();

AA aa = bb;

bb.output();

\*/

AA aa = **new** BB();

BB bb = (BB) aa;

bb.output();

}

}

**interface** AA

{

**public** **void** output();

}

**class** BB **implements** AA

{

**public** **void** output()

{

System.*out*.println("BB");

}

}

### 接口继承接口

Java中还允许一个接口继承于另一个接口，即由父接口派生出子接口，如：

**public**  **interface** 子接口名 **extends** 父接口名

{

成员列表

}

这样的话，甚至可以**使用父接口的引用指向子接口的实现类的对象**。

# 泛型

## 为什么要使用泛型

### 编译时不检查类型的异常

下面的程序，在运行时会出现ClassCastException异常，因为程序试图把一个Integer对象转换为String类型。

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** ListErr {

// 下面的程序，在运行时会出现ClassCastException异常，因为程序试图把一个Integer对象转换为String类型。

@SuppressWarnings({ "unchecked", "rawtypes", "unused" })

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

// 创建一个只想保存字符串的List集合

List strList = **new** ArrayList();

strList.add("Struts2权威指南");

strList.add("基于J2EE的Ajax宝典");

strList.add("轻量级J2EE企业应用实战");

// ”不小心“把一个Integer对象”丢进"了集合

strList.add(5);

**for** (**int** i = 0; i < strList.size(); i++) {

// 因为List里取出的全部是Object，所以必须强制类型转换

// 最后一个元素将出现ClassCastException异常

String str = (String) strList.get(i);

}

}

}

### 需求，使用一个类来完成Float、Sring等类型数据的存储

**class** FloatFoo {

**private** Float foo;

**public** Float getFoo() {

**return** foo;

}

**public** **void** setFoo(Float foo) {

**this**.foo = foo;

}

}

**class** StringFoo {

**private** String foo;

**public** String getFoo() {

**return** foo;

}

**public** **void** setFoo(String foo) {

**this**.foo = foo;

}

}

**解决方案1，定义object类型**

**public** **class** ObjectFoo

{

**private** Object foo;

**public** Object getFoo()

{

**return** foo;

}

**public** **void** setFoo(Object foo)

{

**this**.foo = foo;

}

@SuppressWarnings("unused")

**public** **static** **void** main(String[] args)

{

ObjectFoo foo1 = **new** ObjectFoo();

ObjectFoo foo2 = **new** ObjectFoo();

foo1.setFoo(**new** Boolean(**false**));

Boolean b = (Boolean)foo1.getFoo();

foo2.setFoo(**new** Integer(10));

Integer i = (Integer)foo2.getFoo();

ObjectFoo foo3 = **new** ObjectFoo();

foo3.setFoo(**new** Boolean(**false**));

String str = (String)foo3.getFoo();

}

}

该代码可以解决需求，但是也在运行时会出现ClassCastException异常

## 所谓泛型，就是变量类型的参数化

一个简单的泛型使用示例

**public** **class** GenericFoo<T> {

**private** T foo;

**public** T getFoo() {

**return** foo;

}

**public** **void** setFoo(T foo) {

**this**.foo = foo;

}

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

GenericFoo<Boolean> foo1 = **new** GenericFoo<Boolean>();

GenericFoo<Integer> foo2 = **new** GenericFoo<Integer>();

foo1.setFoo(**new** Boolean(**false**));

foo2.setFoo(**new** Integer(3));

Boolean b = foo1.getFoo();

Integer i = foo2.getFoo();

System.***out***.println(b);

System.***out***.println(i);

//取消注释后会出错，foo1和foo2的类型应经定义好了

//foo1 = foo2;

//同样，在实例化对象时，也可以不指定类型，但是这样就需要进行强制类型转换了

GenericFoo a = **new** GenericFoo();

a.setFoo("hello");

String str = (String)a.getFoo();

System.***out***.println(str);

}

}

object类的equals方法的特点：

equals 方法在非空对象引用上实现相等关系：

自反性：对于任何非空引用值 x，x.equals(x) 都应返回 true。

对称性：对于任何非空引用值 x 和 y，当且仅当 y.equals(x) 返回 true 时，x.equals(y) 才应返回 true。

传递性：对于任何非空引用值 x、y 和 z，如果 x.equals(y) 返回 true，并且 y.equals(z) 返回 true，那么 x.equals(z) 应返回 true。

一致性：对于任何非空引用值 x 和 y，多次调用 x.equals(y) 始终返回 true 或始终返回 false，前提是对象上 equals 比较中所用的信息没有被修改。

对于任何非空引用值 x，x.equals(null) 都应返回 false。

F5 在执行代码时进入方法内部就行查看

F6 直接执行代码的下一句，不会进入方法内部

F8 将代码执行到下一个断点，如果没有断点，则将代码执行到程序结束

**package** com.shengsiyuan;

**import** java.util.ArrayList;

**public** **class** ArrayListTest4

{

**public** **static** **void** main(String[] args)

{

ArrayList list = **new** ArrayList();

list.add(**new** Integer(1));

list.add(**new** Integer(2));

list.add(**new** Integer(3));

list.add(**new** Integer(4));

list.add(**new** Integer(5));

list.add(**new** Integer(6));

/\*

\* 不能将Object[]转换为Integer[]

\*/

Integer[] in = (integer[])list.toArray();

**for**(**int** i = 0; i < in.length; i++)

{

System.*out*.println(in[i].intValue());

}

Object[] in = list.toArray();

**for**(**int** i = 0; i < in.length; i++)

{

System.*out*.println(((Integer)in[i]).intValue());

}

}

}

包装类示例代码：

**public** **class** IntegerTest

{

**public** **static** **void** main(String[] args)

{

**int** a = 10;

Integer b = **new** Integer(a);

Integer c = **new** Integer(10);

System.*out*.println("b = "+b.intValue());

System.*out*.println("c = "+c.intValue());

}

}