# 1. 面向对象2

## 1.1 代码块

代码块用于限定变量的范围，在Java中，还有两个特殊的代码块：

（1）构造代码块：在类的成员位置，用{ }括起来的代码。每次执行构造方法前，都会先执行构造代码块中的代码。

可以把多个构造方法中的共同代码写在构造代码块中，用于对创建的每个对象进行统一初始化，而构造方法只是对用此构造创建的对象初始化。

（2）静态代码块：在类的成员位置，用{ }括起来的代码，并且此段代码用static修饰。

静态代码块是对类进行初始化，**只是在加载类时执行一次。**

代码的执行顺序：静态代码块—构造代码块—构造方法。强调：静态代码块只执行一次，构造代码块每次调用构造方法都执行。

（1）如果想使用静态代码块中的变量，必须将变量提升为类的静态成员，然后外部可以调用。本类的静态方法可访问此变量，实例方法不可以。例子：

|  |
| --- |
| **public class** Student {  **public static** String *str*;  **static** {  *// 静态代码块中初始化*  *str* = **"hello world"**;  }  *// 静态方法中可访问*  **public static void** hello() {  System.***out***.println(*str*);  }  } |

使用：

|  |
| --- |
| **public class** Demo {  **public static void** main(String[] args) {  System.***out***.println(Student.*str*);  Student.*hello*();  }  } |

（2）实例成员变量、构造代码块和构造方法操作同一个变量时的先后顺序：

Java编译后，会把成员变量的定义提升到类的最前面，但是对象成员的初始化都是在构造方法中完成的，即成员变量的赋值和构造代码块都会在构造方法中先执行，最后执行构造方法的内容。至于成员变量的赋值和构造代码块谁先执行，就是按照原本代码的顺序来的。例子：

|  |
| --- |
| **public class** Student {  **int age** = 100;  {  **age** = 200;  }  **public** Student() {  **age** = 300;  }  **public void** showAge() {  System.***out***.println(**age**); *// 实例可访问实例变量*  }  }  如果将构造中的age = 300注释掉，那么输出的就是200，如果再把int age = 100放在构造代码块下面，那么下面输出的就是100了。注意不要重复定义变量，都是在类中定义类变量的，否则外部访问不到。如果在代码块中重新定义了变量，值就不一样了，能访问的只有类中定义的变量。 |

使用：

|  |
| --- |
| **public class** Demo {  **public static void** main(String[] args) {  Student stu = **new** Student();  System.***out***.println(stu.**age**);  stu.showAge();  }  } |

## 1.2 类之间的关系

（1）整体与部分的关系。现在有两个类，一个是地址类，一个是人类。因为人有住址，所以人类中应该有个“住址”属性，该属性的类型就是地址类。这种关系满足“has a”的关系。

例子：

地址类：

|  |
| --- |
| **public class** Address {  **private** String **province**;  **private** String **city**;  **private** String **street**;  **public** String getProvince() {  **return province**;  }  **public void** setProvince(String province) {  **this**.**province** = province;  }  **public** String getCity() {  **return city**;  }  **public void** setCity(String city) {  **this**.**city** = city;  }  **public** String getStreet() {  **return street**;  }  **public void** setStreet(String street) {  **this**.**street** = street;  }  } |

人类：

|  |
| --- |
| **public class** Person {  **private** String **name**;  **private int age**;  **private** Address **addr**; *// 地址信息*  **public** String getName() {  **return name**;  }  **public void** setName(String name) {  **this**.**name** = name;  }  **public int** getAge() {  **return age**;  }  **public void** setAge(**int** age) {  **this**.**age** = age;  }  **public** Address getAddr() {  **return addr**;  }  **public void** setAddr(Address addr) {  **this**.**addr** = addr;  }  } |

（2）继承关系。现在有两个类，一个是动物类，一个是猫类，那么猫类应该继承自动物类，因为猫满足“is a”动物的关系。动物类中是共性，通过继承，有以下好处：

1. 提高代码的复用性，不用写多个类中相同的代码；

2. 提高代码的维护性，关系清晰；

3. 让类与类产生了关系，这是多态的前提。

下面就讲继承。

## 1.2 继承

### 1.2.2 Java中的继承

Java中类的继承只支持单继承，但支持多层继承。即一个类只能继承一个类，允许这个类再被继承，形成继承体系。

继承中，子类只能继承父类非私有的成员；不能继承父类的私有成员和构造方法，但是可以通过super关键字来访问父类的构造方法。

不要为了部分功能去继承。要满足“子类对象 is a 父类对象”的关系。

继承的例子：

（1）Animal类

|  |
| --- |
| **public class** Animal {  *// 有名称属性*  **private** String **name**;  *// 动物有吃的行为*  **public void** eat() {  System.***out***.println(**this**.**name** + **" is eating"**);  }  **public** String getName() {  **return name**;  }  **public void** setName(String name) {  **this**.**name** = name;  }  } |

（2）Cat类

|  |
| --- |
| **public class** Cat **extends** Animal {  *// name属性和eat方法在动物中有了，继承过来的不用再重复*  *// 这里写猫特有的叫的行为*  **public void** miao() {  System.***out***.println(**this**.getName() + **"喵喵叫"**);  }  } |

（3）Demo类中使用

|  |
| --- |
| **public class** Demo {  **public static void** main(String[] args) {  Cat cat = **new** Cat();  *// cat中可以调用父类和本类的非私有成员*  cat.setName(**"小猫"**);  cat.miao();  cat.eat();  }  } |

### 1.2.3 super关键字

super关键字指向父类对象。这个父类对象是从哪里来的。实际上继承中，创建子类对象之前先会创建父类对象，默认子类构造方法的第一行就是super()，就是调用父类的无参构造创建父类对象。

因为默认调用父类的无参构造，所以如果手动写了父类的构造方法（这样父类就没有了默认的无参构造），那么需要在子类的构造方法第一行手动用super关键字调用父类对应的构造方法。

总结this和super的用法：

（1）调用成员变量

this.成员变量 调用本类的成员变量

super.成员变量 调用父类的成员变量

（2）调用构造方法

this(…) 调用本类的构造方法

super(…) 调用父类的构造方法

（3）调用成员方法

this.成员方法() 调用本类的成员方法

super.成员方法() 调用父类的成员方法

### 1.2.4 方法重写(override)

方法重写就是子类重写父类的方法，这样当子类对象调用该方法时，会表现出子类的行为，而父类对象调用该方法时，表现的还是父类的行为。

子类方法与父类方法同名，自动重写。无法重写父类的私有方法

子类重写父类的方法时，访问权限不能更低，最好一致。而且返回值类型必须是父类函数的返回值类型或该返回值类型的子类。不能返回比父类更大的数据类型。

## 1.3 instanceof关键字

instanceof属于比较运算符，该关键字用来判断一个对象是否是指定类的对象。返回的结果是boolean类型。用法是“对象 instanceof 类”。要求是该对象的类必须和instanceof后的类有继承关系，否则报错。

|  |
| --- |
| **public class** Demo {  **public static void** main(String[] args) {  Cat cat = **new** Cat();  Animal animal = **new** Animal();  System.***out***.println(cat **instanceof** Cat);  System.***out***.println(cat **instanceof** Animal);  System.***out***.println(animal **instanceof** Cat);  *// 不能用 cat instanceof System，因为System类和Cat类没有继承关系。*  String[] strs = {**""**};  System.***out***.println(strs **instanceof** String[]);  }  } |

## 1.4 final关键字

final的引入：现在需要访问类中的变量，但是不希望这个值被改变，则用final修饰一个这个类的静态变量即可，即：

public static final NUM = 100.

比如在Math类中的PI变量，通过Math.PI访问，但是不能改变他的值。

关于final的作用：

（1）final修饰类：该类不能被继承。

（2）final修饰方法，该方法不能被重写。

（3）final修饰变量，该变量不能被重新赋值。常来定义常量。

（4）final修饰成员变量

若修饰基本类型，则其值不能改变

若修饰引用类型：引用类型的地址值不能改变。但该对象的属性可以改变，因为他在堆内存中。也就是可以：

final Person p = new Person();不能p = new Person()，可以p.setName(“张三”)

（5）final修饰形参，则在该方法中不能修改这个形参。比如某个方法中只需要遍历传递进来的数组，那么为了数据安全性，则可以将这个数组形参加上final关键字。

（6）final变量的初始化

如果用静态常量，则必须是声明时便赋值，即：static final int a = 9;否则没有其他时机赋值

如果是非静态常量，则在每个构造方法里面赋值（或之前已经赋好值）。

## 1.5 多态

多态就是一个对象可以有多种状态。实现多态的前提：

（1）类之间有继承关系；

（2）子类重写父类的方法。

然后，用父类类型对象接收子类创建的对象（父 f = new 子()），这个就是多态，因为其特点是：

（1）由于是父类类型接收的，“名义”上是父类类型，表现的是父类的属性。

（2）但是该对象“本质”是子类对象，由于子类重写了父类的方法，那么虽然现在的父类类型，当调用这个被重写的方法时，还是会调用被重写的方法。并且，该对象可被强制转换成子类类型对象。

例子：

（1）父类

|  |
| --- |
| **public class** Fu {  **public int num** = 100; *// 成员变量*  *// 实例方法*  **public void** show() {  System.***out***.println(**"Show Fu"**);  }  *// 静态方法*  **public static void** showStatic() {  System.***out***.println(**"Show Fu Static"**);  }  } |

（2）子类

|  |
| --- |
| *// 继承父类Fu*  **public class** Zi **extends** Fu {  *// 成员变量*  **public int num** = 200;  *// 重写实例方法*  **public void** show() {  System.***out***.println(**"Show Zi"**);  }  *// 静态方法*  **public static void** showStatic() {  System.***out***.println(**"Show Zi Static"**);  }  } |

（3）多态使用

|  |
| --- |
| **public class** Demo {  **public static void** main(String[] args) {  Fu f = **new** Zi();  System.***out***.println(f.**num**); *// 输出成员变量* f.show(); *// 实例方法* f.*showStatic*(); *// 静态方法* } } |

说明：

上面例子中，子类“重写”父类的成员变量、方法和静态方法。然后创建子类，但是用父类接收。

对于变量，并没有“重写变量”这个概念，所以用父类调用，还是会找到父类的num也就是1000.

对于实例方法，子类已经重写了，所以表现出子类的方法。

对于静态方法，因为静态和类相关，不算是重写。所以还是调用父类的方法。

多态常见的是“转型”问题：

（1）向上转型： Fu f = new Zi();

（2）向下转型： Zi z = (Zi)f; //要求f必须是能够转换为Zi的。

## 1.6 抽象类

如果一个类中的方法是抽象的，不知如何实现，那么可将此方法和此类定义成抽象的，不实现抽象方法（即不写方法体），然后，继承该类的具体实现类可以重写该方法，完成具体实现。用abstract关键字定义抽象方法和抽象类。

使用abstract应该注意：

（1）抽象类和抽象方法用abstract关键字修饰；

（2）有抽象方法的类必须定义为抽象类，但是抽象类中不一定有抽象方法。

（3）抽象类由于有方法没有具体实现，因此不能直接实例化。抽象类也可用构造方法进行数据的初始化。

（4）抽象类中也可以有非抽象的成员，比如变量、方法等。

（5）抽象类的实例化是靠具体的子类实现的。是多态的方式。例如：Animal a = new Cat();

（6）抽象类的子类如果还是一个抽象类，则可以不实现抽象方法；如果是一个具体类，则必须实现（重写）所有的抽象方法。

（7）abstract不能和private、final和static共存，因为这些关键字与重写方法冲突（这些关键字使得外界不能重写方法）。

下面就是一个例子，Shape（形状）类中不知如何实现求面积的方法，因此设计为抽象的，然后他的具体实现（矩形、圆形）可以进行具体实现，然后可以用多态的方式。

（1）Shape类

|  |
| --- |
| **public abstract class** Shape {  *// 定义抽象的方法，求面积，不要写实现，就是不写大括号及其里面的内容* **public abstract double** area(); } |

（2）Rect类

|  |
| --- |
| **public class** Rect **extends** Shape {  *// 定义宽高属性* **private double width**;  **private double height**;  *// 在构造中初始化宽高* **public** Rect(**int** width, **int** height) {  **this**.**width** = width;  **this**.**height** = height;  }   *// 实现求面积的方法* **public double** area() {  **return this**.**width** \* **this**.**height**;  }   **public double** getWidth() {  **return width**;  }   **public void** setWidth(**double** width) {  **this**.**width** = width;  }   **public double** getHeight() {  **return height**;  }   **public void** setHeight(**double** height) {  **this**.**height** = height;  } } |

（3）Circle类

|  |
| --- |
| **public class** Circle **extends** Shape {  *// 定义半径* **private double radius**;  *// 构造* **public** Circle(**double** radius) {  **this**.**radius** = radius;  }  *// 实现求面积的方法* **public double** area() {  **return** Math.***PI*** \* **this**.**radius** \* **this**.**radius**;  } } |

（4）使用的Demo类

|  |
| --- |
| **public class** Demo {  **public static void** main(String[] args) {  *// 多态创建矩形* Shape rect = **new** Rect(12, 4);  System.***out***.println(rect.area());  *// 多态创建圆形* Shape circle = **new** Circle(2);  System.***out***.println(circle.area());  } } |

当然也可以不用多态，直接用本类类型接收。

## 1.7 接口

接口，就是用来扩展功能的，可以弥补Java中单继承的缺点。比如现在有带橡皮擦的铅笔，如果此类继承了橡皮类，就能有erase的功能，如果此类继承了铅笔类，就有了write的功能。但是由于Java的单继承的，因此不能同时实现上述两个功能。所以需要引入接口。

接口中只需声明抽象的方法，如果某类实现了接口，那么就要实现接口中所有的方法。然后通过多态方式创建实例，调用方法。因此接口也是实现多态的一种方法。

注意：

（1）用关键字interface定义接口，例如：

interface 接口名 {}

（2）类实现接口用implements关键字，实现接口的类一般以Impl作为后缀。

class 类名 implements 接口名 {}

（3）接口不能直接被实例化，而是按照多态的方式来实例化。

（4）接口的子类，如果是抽象类，不需要写具体实现；如果是具体类，则要重写接口中的所有方法。

（5）接口没有构造方法。

（6）接口中只能有抽象方法，不能有具体的方法。接口中方法的默认修饰符是public abstract，可以省略，比如直接写void run();等。但是为了加强意识，可以主动加上。

（7）接口没有构造方法。

（8）接口中可定义属性，默认的修饰符是public static final，即静态常量，必须定义的时候赋值。这些常量可以通过“接口名.属性名”来使用。

（9）一个类可以实现多个接口。

下面是带橡皮的铅笔的例子。

（1）Erase接口

|  |
| --- |
| **public interface** Erase {  **void** erase(); *// 擦除的功能* } |

（2）Write接口

|  |
| --- |
| **public interface** Write {  **void** write(); *//写的功能* } |

（3）PencilWithErase类实现上述两个接口

|  |
| --- |
| **public class** PencilWithErase **implements** Erase, Write {  **private** String **name**; *//名称* **public** PencilWithErase(String name) {  **this**.**name** = name;  }   *// 实现擦除和写的功能* **public void** erase() {  System.***out***.println(**this**.**name** + **"is erasing..."**);  }  **public void** write() {  System.***out***.println(**this**.**name** + **"is writing..."**);  } } |

（4）在Demo中多态使用

|  |
| --- |
| **public class** Demo {  **public static void** main(String[] args) {  *// 多态实现擦除和写* Erase erase = **new** PencilWithErase(**"带橡皮的铅笔"**);  erase.erase();   Write write = **new** PencilWithErase(**"带橡皮的铅笔"**);  write.write();  } } |

## 1.8 Java的继承体系