Java面向对象编程(下)

讲授：李钦坤

Java面向对象编程(下)

- 4.1 类的继承
- 4.2 super关键字
- 4.3 访问控制
- 4.4 方法重写
- 4.5 static关键字
- 4.6 final关键字
- 4.7 多态
- 4.8 抽象类
- 4.9 接口
- 4.10 嵌套类

4.1 继承

- 继承关系：两个类之间是“is-a”(是一个)的关系。
- 继承的特点：
  - 子类继承了父类(父类,基类)的所有属性和方法(但不包括构造器)，并且可以为自己增添新的属性和方法。
- 继承可提高代码的重用性，使用extends关键字来实现。
- Java只支持单继承：即一个子类只能有一个父类。但一个父类可以派生出多个子类。

示例

```
class Employee {
    private String name;                    //姓名
    private double salary = 2000.0;     //薪水
    public Employee(String name, double salary){
        this.name = name;
        this.salary = salary;
    }
    public Employee(){}
    public double getSalary(){        return salary;    }
}
class Manager extends Employee{
    private double bonus;    //奖金
    public void setBonus(double bonus){        this.bonus = bonus;    }
}
public class InheritanceTest {
    public static void main(String [] args){
        Manager manager = new Manager();
        manager.getSalary();
    }
}
```

子类的实例化过程

- 调用子类构造方法创建一个对象时：
  - 会先调用父类构造方法对父类中的属性进行初始化，然后才调用自己的构造方法对自己的属性来进行初始化
  - 如果子类构造方法没有显式调用父类构造方法，那么会调用父类的默认构造方法。
  - 如果父类没有默认构造方法，而且子类构造方法又没有显式调用父类的其它构造方法，那么编译将报错。

4.2 super

- 要在子类构造方法中显式调用父类的构造方法，则在子类构造方法的第一条语句用关键字 super来调用。
- 语法为：
  - super()； //显式调用父类的无参构造方法
  - super(实参列表)； //显式调用父类的带参构造方法

示例

```
class Employee {
    private String name;                    //姓名
    private double salary = 2000.0;     //薪水
    public Employee(String name, double salary){
        this.name = name;
        this.salary = salary;
    }
    public Employee(){}
    public double getSalary(){        return salary;    }
    public void displayInfo(){        System.out.println(“name=” + name);   }
}
class Manager extends Employee{
    private double bonus;    //奖金
    private String position;   //职位
    public Manager(String name, double salary, String position){
        super(name, salary);
        this.position = position;
    }
    public void setBonus(double bonus){        this.bonus = bonus;    }
}
```

super的第二种用法

- super还可用来显式调用父类的普通方法。
  - 语法：super.方法名(实参列表);

```
class Employee {
    …
    public void displayInfo(){        System.out.println(“name=” + name);    }
}

class Manager extends Employee{
    private double bonus;    //奖金
    private String position;   //职位

    public Manager(String name, double salary, String position){
        super(name, salary);
        this.position = position;
        super.displayInfo();
    }

    public void setBonus(double bonus){        this.bonus = bonus;     }
}
```

4.3 访问修饰符

位置	private	默认	protected	public
同一个类	是	是	是	是
同一个包内的类		是	是	是
不同包内的子类			是	是
不同包并且不是子类				是

- 访问修饰符可以修饰类、成员变量、方法
  - private： 本类可见 （经常用来修饰类中的属性）
  - 默认：本类、本包可见（不建议使用）
  - protected：对本类、所有子类和本包可见 （用法特殊）
  - public：对一切可见（经常用来修饰类，修饰类中的方法）
- 对于外部类的访问修饰，只可以是public或 默认 的。

示例

```
class T {
        private int i = 10;
        int j = 100;
        protected int k = 1000;
        public int m = 100000;

}

public class AccessTest {
        public static void main(String [] args) {
                T t = new T();
                System.out.println(t.i);
                System.out.println(t.j);
                System.out.println(t.k);
                System.out.println(t.m);
        }
}
```

4.4 方法的重写(override)

- 在子类中可以根据需要对从父类中继承来的方法进行重写。
- 重写方法必须和被重写方法
  - 具有相同的方法名称、参数列表和返回值类型
- 注意：
  - 子类中的重写方法的访问范围不能被缩小
  - 静态方法只能重写父类的静态方法。

示例

```
class Employee {
    private String name;                    //姓名
    private double salary = 2000.0;     //薪水
    public Employee(String name, double salary){
        this.name = name;
        this.salary = salary;
    }
    public double getSalary(){        return salary;    }
}

class Manager extends Employee{
    private double bonus;    //奖金
    private String position;   //职位
    public Manager(String name, double salary, String position){
        super(name, salary);
        this.position = position;
    }
    public void setBonus(double bonus){        this.bonus = bonus;    }
    public double getSalary(){
         return super.getSalary() + bonus;
    }
}
```

4.5 static关键字

- static声明的成员变量为静态成员变量：
  - 它是该类的公用变量，在第一次使用时被初始化，对于该类的所有对象来说，static成员变量只有一份。也说成“类变量”。
- static声明的方法为静态方法：
  - 该方法独立于类的实例，所以也叫“类方法”。
  - 静态方法中只能调用本类中其它的静态成员(变量和方法)。
  - 静态方法中不能使用this和super关键字。
- 静态成员可以通过类名(不需要实例化)或类的实例去访问。

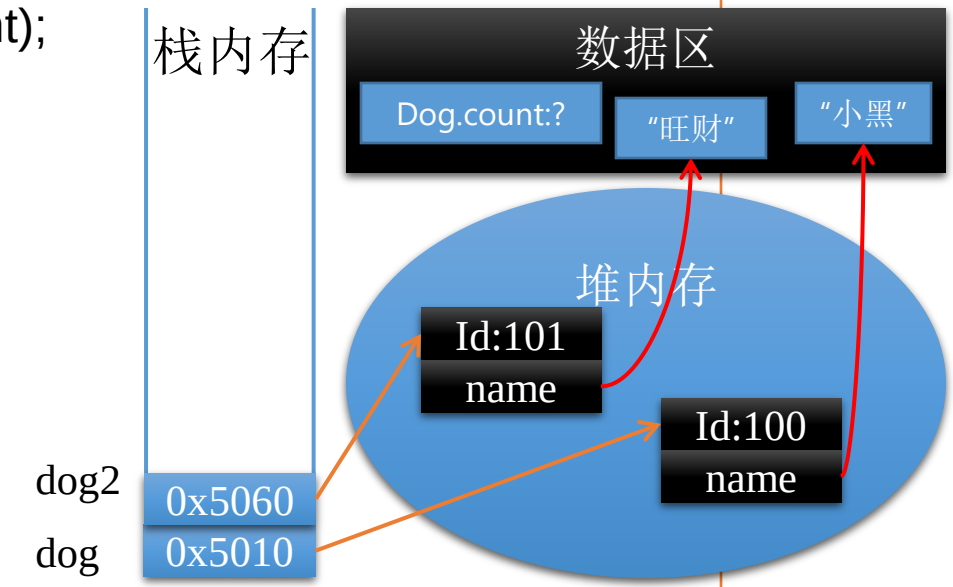
static示例

```
public class Dog{
    private static int count = 100;
    private int id;
    private String name;

    public Dog(String name){
        this.name = name;
        id = count++;
    }
    public void displayInfo(){
        System.out.print(“名字是:” + name + 编号是" + id);
    }
    public static void currentCount(){
        System.out.println("当前编号是" + count);
    }

    public static void main(String[] args){
        Dog.currentCount();
        Dog dog = new Dog("小黑");
        dog.displayInfo();

        Dog dog2 = new Dog(“旺财”);
        dog2.displayInfo();
        Dog.currentCount();
    }
}
```



静态代码块

- 在类中可以使用不包含在任何方法中的静态代码块(static block)，当类被JVM载入时，静态代码块被执行，且只被执行一次。
- 静态代码块经常用来初始化类的静态成员变量。

```
public class Student{
    private static String country;
    private String name;
    static {
        country = "中国";
        System.out.println("Student类已经加载");
    }
    public Student(String name){
        this.name = name;
    }
    public void displayInfo(){
        System.out.println("我的名字是:" + name + ",国家是:" + country);
    }
    public static void main(String[] args){
        //Student stu = new Student("张三");
        //stu.displayInfo();
    }
}
```

4.6 final关键字

- final修饰的变量(成员变量或局部变量)的值不能改变，即等同于常量，在定义时就必须要进行初始化，之后就再也不改变它的值了。
  - final修饰方法的参数叫最终参数。调用这个方法时给最终参数赋值之后，在方法体内再也不能改变它的值。
- final修饰的方法不能被子类重写。
- final修饰的类不能被继承。

```
public class TestFinal {
	public static void main(String[] args) {
		T t = new T();
		//t.i = 100;
	}
}
final class T {
	final int i = 10;
	public final void m() {	//j = 11;	}
}
class TT extends T {}
```

4.7 多态

- 同一类事物的多种存在形态。

4.7.1 对象多态性

- 一个父类变量可以“指向”其子类的对象。
- 一个父类变量不可以访问其子类对象新增加的成员。
- 可以使用instanceof关键字来判断该对象变量所“指向”的对象是否属于该类。
  - 语法：对象变量名 instanceof 类名(或接口名)

- 子类的对象可以直接当作父类的对象使用，称作向上转型。它是自动进行的。
- 从父类的对象到子类的对象的转换叫向下转型，向下转型要用强制类型转换。

示例

```
class Animal {
  private String name;
  Animal(String name) {
      this.name = name;
  }
  public String getName(){
    return name;
  }
}
class Cat extends Animal {
  private String eyesColor;
  Cat(String n,String c) {
      super(n); eyesColor = c;
   }
  public String getEyesColor(){
    return eyesColor;
  }
}
class Dog extends Animal {
  private String furColor;
  Dog(String n,String c) {
      super(n); furColor = c;
  }
  public String getFurColor() {
    return furColor;
  }
}
```

```
public class CastingTest{
    public static void main(String args[]){
        Animal a = new Animal("动物");
        Cat c = new Cat("猫","black");
        Dog d = new Dog("狗","yellow");

        System.out.println(a  instanceof  Animal);
        System.out.println(c  instanceof  Animal);
        System.out.println(d  instanceof  Animal);
        System.out.println(a  instanceof  Cat);

         //向上转型
         Animal an = new Dog("旺财","yellow");
        System.out.println(an.getName());
        System.out.println(an.getFurColor());  //error!
        System.out.println(an  instanceof  Animal);   //true
        System.out.println(an  instanceof  Dog);  //true

        //向下转型,要加强制转换符
        //-- 为了安全起见，要加 instanceof 判断
        Dog d2 = (Dog)an;
        //Cat c2 = (Cat)an;
        System.out.println(d2.getFurColor());
    }
}
```

4.7.2 多态方法

- 多态方法可以提高代码的可维护性和可扩展性。
- 发生多态的条件
  - 要有继承
  - 要有重写
  - 要有向上转型（父类变量指向子类对象）
  - 父类变量调用被子类重写的方法

示例

```
class Animal {
  private String name;
  public Animal(String name) {
      this.name = name;
  }
  public void enjoy(){
    System.out.println("叫声.....");
  }
}
class Cat extends Animal {
  private String eyesColor;
  public Cat(String n) {super(n);}

  public void enjoy(){
    System.out.println("喵~喵~");
  }
}
class Dog extends Animal {
  private String furColor;
  public Dog(String n) {super(n);}

  public void enjoy(){
    System.out.println("汪~汪~");
  }
}
```

```
class Lady{
  private String name;
  private Animal pet;
  public Lady(String name, Animal pet) {
    this.name = name;
    this.pet = pet;
  }
  public void myPetEnjoy(){
     pet.enjoy();
  }
}

public class DynamicBindingTest {
    public static void main(String args[]){
        Cat c = new Cat("catname","blue");
        Lady l = new Lady("张女士", d);
        l.myPetEnjoy();
    }
}
```

4.8 抽象类

- 用abstract修饰的方法叫抽象方法。
  - 抽象方法形式：(没有方法体)
    - [访问修饰符] abstract 返回类型 方法名(参数列表);
- 用abstract修饰的类叫抽象类。
  - 声明抽象类语法：
    - [访问修饰符] abstract class 类名{…… }
- 抽象类不能被实例化。
- 含有抽象方法的类必须被声明抽象类。
- 抽象类的子类必须重写所有的抽象方法后才能被实例化，否则这个子类也要声明成抽象的。

示例

```
abstract class Shape {  //形状类
 protected double length;    //长
 protected double width;    //宽
    public Shape(double length, double width){
        this.length = length;  this.width = width;
    }
    public abstract double area();    //计算面积
}
class Rectangle extends Shape {   //矩形
    Rectangle(final double num, final double num1) {        super(num, num1);    }
    public double area() {        return length * width;    }
}
class Triangle extends Shape{    //三角形
    Triangle(final double num, final double num1) {        super(num, num1);    }
    public double area() {        return length * width/2;    }
}
public class TestAbstract{
    public static void main(String[] args){ … }
}
```

思考

- 抽象类中可不可以没有抽象方法？
- 有抽象方法的类可以可以不是抽象类？

4.9 接口(interface)

- 接口就是某个事物对外提供的一些功能的申明 。
- 可以利用接口实现多态，同时也弥补Java单一继承的弱点
- 使用interface关键字定义接口。
- 一般使用接口声明方法或静态最终变量(常量)，接口中的方法只能是声明，不能是具体的实现。
- 接口中的任何方法都自动置为public的，属性也总是public static final的。
- 接口没有构造方法，所以不能被实例化(不能用来创建对象)。
- 从JavaSE 8开始，接口中也可以声明默认方法与静态方法。默认方法使用default修饰，可以通过实现类的对象调用。静态方法使用static修饰，可以通过接口名调用

创建接口的示例

```
//方法接口
public interface Runner{
    public void run();
}

//定义常量的接口
public interface Constants{
  public static final int COLOR_RED = 1;
  public static final int COLOR_GREEN = 2;
  public static final int COLOR_BLUE = 3;
}
```

实现接口

- 用关键字 implements 实现接口。如：
  - class Car implements Runner
- 每个类只能有一个父类，但可以实现多个接口。如果实现多个接口，则用逗号隔开接口名称，如下所示：
  - class Car implements Runner, Constants
- 一个类实现了一个接口，它必须实现接口中定义的所有方法，否则该类必须声明为抽象类。
- 接口可以继承自其它的接口，并添加新的常量和方法。接口支持多重继承。

示例

```
class Car implements Runner,Constants{  //实现两个接口
	public void run(){
		System.out.println("车颜色是:" + COLOR_RED);
		System.out.println("用四个轮子跑...");
	}
}

interface Animal extends Runner{   //接口的继承
	void breathe(); //呼吸
}

class Fish implements Animal{
	public void run(){
		System.out.println("颜色是:" + COLOR_BLUE);
		System.out.println("游啊游...");
	}
	public void breathe(){
		System.out.println("冒气泡来呼吸");
	}
}
```

4.10 嵌套类

- 声明在类的内部的类称之为嵌套类（nested class）
- 定义语法：

```
[public] class OuterClass{
    ...
    [public|proteceted|private] [static] class NestedClass{
        ...
    }
}
```

- 分类：
  - 内部类：非静态嵌套类
  - 静态嵌套类：用static修饰的嵌套类
- 嵌套类在编译后会生成OuterClass$NestedClass.class类文件

4.10.1 内部类

- 内部类作为外部类的一个成员存在，与外部类的成员变量、成员方法并列。

```
class Outer {
    private int outer_i = 100;
    private int j = 123;
    public void test() {       System.out.println("Outer:test()");    }
    public void accessInner(){
        Inner inner = new Inner();//外部类中使用内部类也需要创建出它的对象
        inner.display();
    }
    public class Inner {
        private int inner_i = 100;
        private int j = 789;     //与外部类某个属性同名
        public void display() {
            //内部类中可直接访问外部类的属性
            System.out.println("Inner:outer_i=" + outer_i);
            test();      //内部类中可直接访问外部类的方法
            //内部类可以用this来访问自己的成员
            System.out.println("Inner:inner_i=" + this.inner_i);
            System.out.println(j);  //访问的是内部类的同名成员
            //通过"外部类.this.成员名"来访问外部类的同名成员
            System.out.println(Outer.this.j);
        }
    }
}
```

```
public class MemberInnerClassTest {
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.test();
        outer.accessInner();
        //在外部类以外的地方创建内部类的对象
        Outer.Inner inner = outer.new Inner();
        inner.display();
    }
}
```

4.10.2 静态嵌套类

- 静态嵌套类中可声明static成员或非静态成员，但只能访问外部类中的静态成员。

```
[public ] class OuterClass{
    ...
    static class StaticNestedClass { //静态嵌套类
        ...
    }
}
```

4.10.3 方法类

- 方法类（局部内部类）：在方法体内声明的类
  - 可访问它所在方法中的final参数和final局部变量
  - 由于线程与并发的原因，局部内部类仅能方法中的final参数和final局部变量。

```
public class Outer{
    public void test(){
        final int x = 9;//可以访问final修饰的变量
        int y = 10;//jdk8.0后 可以访问只赋值过一次的变量
        //y = 12;//非final修饰的变量，如果改变它的值，则不能被方法类访问

            class FunctionInner{//方法类(局部内部类)
                    public void show(){      System.out.println(x+" "+y);  }
}
    FunctionInner f = new FunctionInner();//创建方法类对象
    f.show();//调用方法类中的方法
}
```

- 说明：从JavaSE 8开始，方法类也能访问非final的局部变量，但前提是该局部变量只能被赋值一次，因此从效果上说，该变量与final的等价的。

4.10.4 匿名内部类

匿名内部类：没有声明名称的内部类

匿名类与方法类相同，仅能访问方法中final的局部变量。JavaSE 8以后也可以访问final等价的局部变量。

```
[public ] class OuterClass{
    ...
    new 已经存在的类名或接口名 () {
        ...
    }
}
```

```
public class AnnonymoseInnerClassTest {
    public static void main(String[] args) {
        (new AClass("redhacker") {
            public void print() {  //对父类的print方法进行覆盖
                System.out.println("the anonymose class print");
                super.print();  //调用父类中的print方法
            }
        }).print();  //调用覆盖后的print方法
    }
}
class AClass {
    private String name;
    AClass(String name) {        this.name = name;}
    public void print() {
        System.out.println("SuperClass:The name = " + name);
    }
}
```

说明

- 方法类和匿名内部类最常见的用处就是实现一个接口，而这个接口通常都只会使用一次，所以不用专门用一个类去实现它。
- 比起方法类，匿名类更加简洁。但当类体较长时，这时候就不适合使用匿名类了，而是应该使用方法类。

λ(Lambda) 表达式

简单介绍 λ(Lambda) 表达式

- 只能适用于函数(型)接口(functional interface),接口中只有一个抽象方法
- 语法结构：(形参列表) -> { 要实现对应抽象方法的代码块}

可以根据不同的情况简化该结构

- (参数列表)部分：有多个参数时，可以(int i，int j)、(i,j)；
- 一个参数，(i)、i->；
- 无参数，()-> 注意:不能省略()

{代码块}部分：多个语句时，方法体；void方法，方法体；有返回值的单一语句，

例：int getX()。其λ表达式为 ()->{return 0;} 或 ()->0;

int add(int i，int j)。其λ表达式为(i,j)->{return i+j;} 或 (i,j)->i+j;

总结

- Extends
- this
- super
- 访问修饰符
- Override
- static
- final
- 多态 polymorphism
  - 向上转型upcasting/向下转型downcasting
  - 动态绑定dynamic binding
- abstract
- interface implements