

# 动态绑定/多态 polymorphism;dynamic bind;late bind;runtime bind

- 动态绑定是指“在执行期间（而非编译期）判断所引用对象的实际类型，根据其实际的类型调用其相应的方法

TestPolymoph/Test.java

- 上面例子中，根据 Lady 对象的成员变量 pet 所引用的不同的实际类型而调用相应的 enjoy 方法

```
class Bird extends Animal {
    private String featherColor;
    Bird(String n,String f) {
        super(n); featherColor = f;
    }
    public void enjoy() {
        System.out.println("鸟叫声.....");
    }
}

public class Test {
    public static void main(String args[]){
        Lady l3 = new Lady
            ("l3",new Bird("birdname","green"));
        l3.myPetEnjoy();
    }
}
```

多态的存在要有3个必要条件：

要有继承，要有重写，父类引用指向子类对象

# 总结

- 1.要有继承
- 2.要有重写
- 3.父类引用要指向子类对象

# 抽象类

- 用abstract关键字来修饰一个类时，这个类叫做抽象类；用abstract来修饰一个方法时，该方法叫做抽象方法。
- 含有抽象方法的类必须被声明为抽象类，抽象类必须被继承，抽象方法必须被重写。如果重写不了，应该声明自己为抽象。
- 抽象类不能被实例化。
- 抽象方法只需声明，而不需实现。

抽象方法就是用来被继承的

抽象方法没有被实现，就是作一个标识符。子类去实现所以必须被重写

TestPolymorph/TestAbstract.java

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

# Final关键字

- final的成员变量的值不能够被改变
- final的方法不能够被重写
- final的类不能够被继承

如果你想定义你的类不被其他人继承，使用**final**

**final**可以用来修饰变量，可以用来修饰方法，可以用来修饰类。

TestFinal.java

# 接口

在这个抽象类里面，所有的方法都是抽象的方法，并且这个抽象类的属性，也就是成员变量是`public static final`

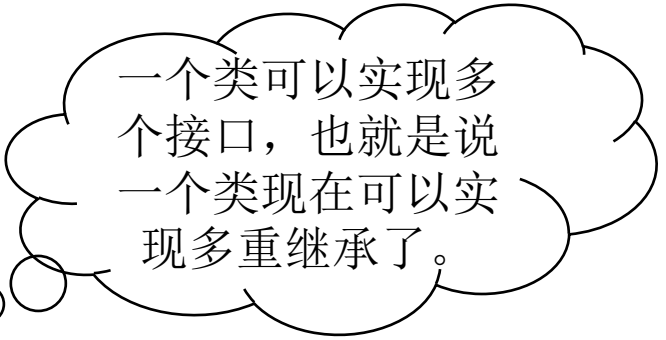
- 接口（**interface**）是抽象方法和常量值的定义的集合。
- 从本质上讲，**接口是一种特殊的抽象类**，这种抽象类中只包含常量和方法的定义，而没有变量和方法的实现。
- 接口定义举例：

为了修正C++里多继承的时候容易出问题的地方：多继承的多个父类之间如果有相同的成员变量，这个引用起来会相当的麻烦，并且运行的时候会产生各种各样的问题。

所以把这个接口的所有的成员变量全都改成`static final`

```
public interface Runner {  
    public static final int id = 1;  
  
    public void start();  
    public void run();  
    public void stop();  
}
```

# 接口特性



一个类可以实现多个接口，也就是说一个类现在可以实现多重继承了。

- 接口可以多重实现；
- 接口中声明的属性默认为 **public static final** 的；也只能是 **public static final** 的；
- 接口中只能定义抽象方法，而且这些方法默认为 **public** 的、也只能是 **public** 的；
- 接口可以继承其它的接口，并添加新的属性和抽象方法。

# 接口

- 多个无关的类可以实现同一个接口。
- 一个类可以实现多个无关的接口。
- 与继承关系类似，接口与实现类之间存在多态性。
- 定义Java类的语法格式：

```
< modifier> class < name> [extends < superclass>]  
[implements < interface> [,< interface>]* ] {  
    < declarations>*  
}
```



# 接口 举例

```
interface Singer {  
    public void sing();  
    public void sleep();  
}  
class Student implements Singer {  
    private String name;  
    Student(String name) {  
        this.name = name;  
    }  
    public void study(){  
        System.out.println("studying");  
    }  
    public String getName(){return name;}  
    public void sing() {  
        System.out.println("student is singing");  
    }  
    public void sleep() {  
        System.out.println("student is sleeping");  
    }  
}
```



# 接口 举例

```
interface Singer {
    public void sing();
    public void sleep();
}
interface Painter {
    public void paint();
    public void eat();
}

class Student implements Singer {
    private String name;
    Student(String name) {this.name = name;}
    public void study(){System.out.println("studying");}
    public String getName(){return name;}
    public void sing() {System.out.println
                        ("student is singing");}
    public void sleep() {System.out.println
                        ("student is sleeping");}
}
```

# 接口 举例

```
class Teacher extends Object implements Singer,Painter {  
    private String name;  
    public String getString() {  
        return name;  
    }  
    Teacher(String name){this.name = name;}  
    public void teach(){System.out.println("teaching");}  
    public void sing(){System.out.println  
        ("teacher is singing");}  
    public void sleep(){System.out.println  
        ("teacher is sleeping");}  
    public void paint(){System.out.println  
        ("teacher is painting");}  
    public void eat(){System.out.println  
        ("teacher is eating");}  
}
```

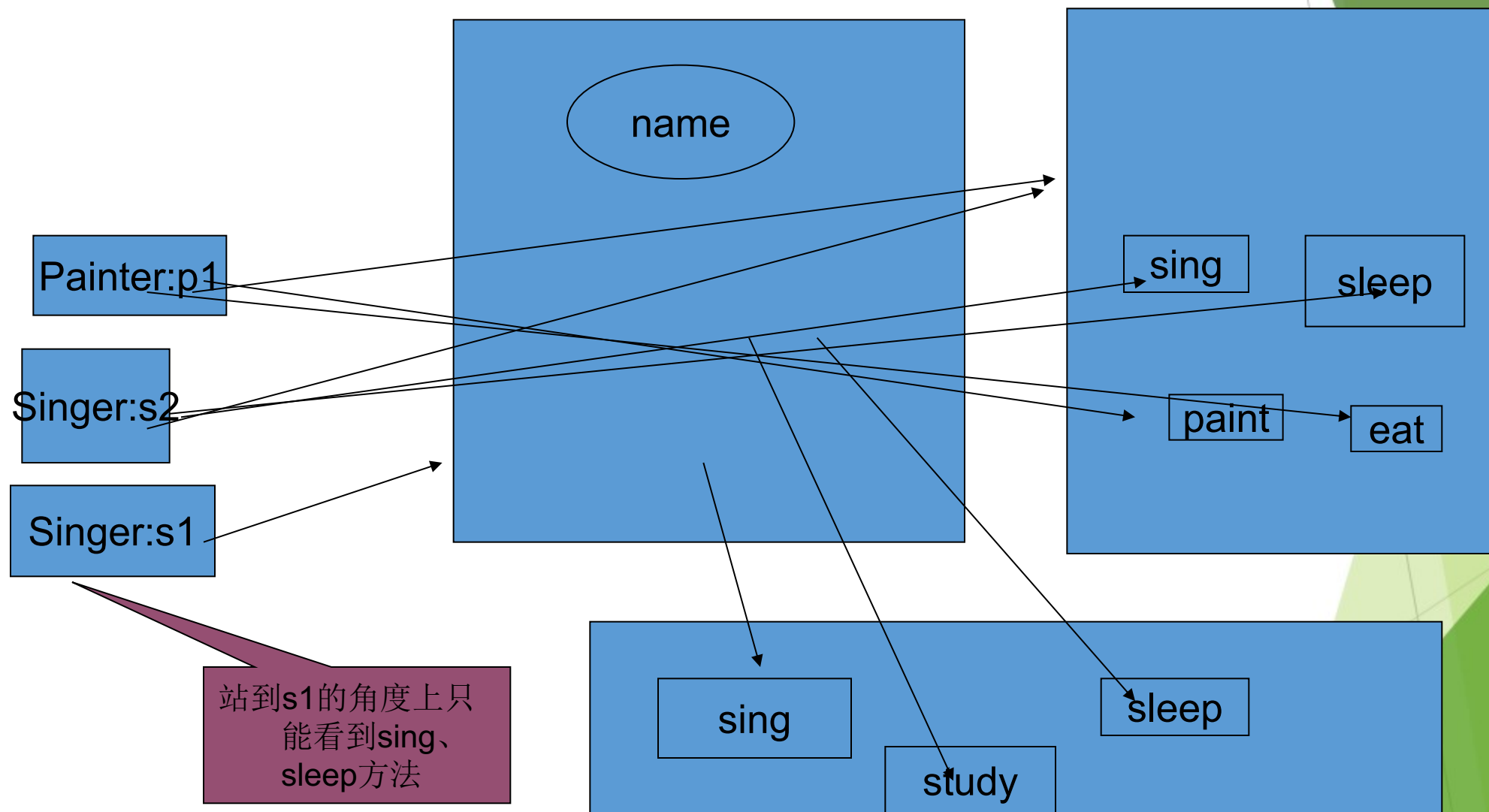
# 接口 举例

```
public class Test {  
    public static void main(String args[]){  
        Singer s1 = new Student("le");  
        s1.sing(); s1.sleep();  
        Singer s2 = new Teacher("steven");  
        s2.sing(); s2.sleep();  
        Painter p1 = (Painter)s2; //也可以当作Painter来看  
        p1.paint(); p1.eat();  
    }  
}
```

➤ 输出结果:

```
student is singing  
student is sleeping  
teacher is singing  
teacher is sleeping  
teacher is painting  
teacher is eating
```

test/Test.java



# 产生常量群

```
interface Months {
```

```
    int
```

```
        JANUARY = 1, FEBRUARY = 2, MARCH = 3,
```

```
        APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
```

```
        AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
```

```
        NOVEMBER = 11, DECEMBER = 12;
```

```
}
```

- 采用如下方法取用其值

- Months.JANUARY

# 数据成员初始化

```
import java.util.*;

interface RandVals {
    Random rand = new Random();
    int randomInt = rand.nextInt(10);
    long randomLong = rand.nextLong() * 10;
    float randomFloat = rand.nextLong() * 10;
    double randomDouble = rand.nextDouble() * 10;
}

public class Test {
    public static void main(String[] args) {
        System.out.println(RandVals.randomInt);
        System.out.println(RandVals.randomLong);
        System.out.println(RandVals.randomFloat);
        System.out.println(RandVals.randomDouble);
    }
}
```

RandVals/Test.java

# 接口用法总结

- 通过接口可以实现不相关类的相同行为，而不需要考虑这些类之间的层次关系。(就像人拥有一项本领)
- 通过接口可以指明多个类需要实现的方法。(描述这项本领的共同接口)
- 通过接口可以了解对象的交互界面，而不需了解对象所对应的类。
- 使用接口？还是抽象类？
  - interface同时赋予了接口和抽象类的好处
  - 如果父类可以不带任何函数定义，或任何成员变量，那么优先使用接口。