# 面向对象编程

## 本节目标

- 继承
- 组合
- 多态

## 1 继承

### 1.1 背景

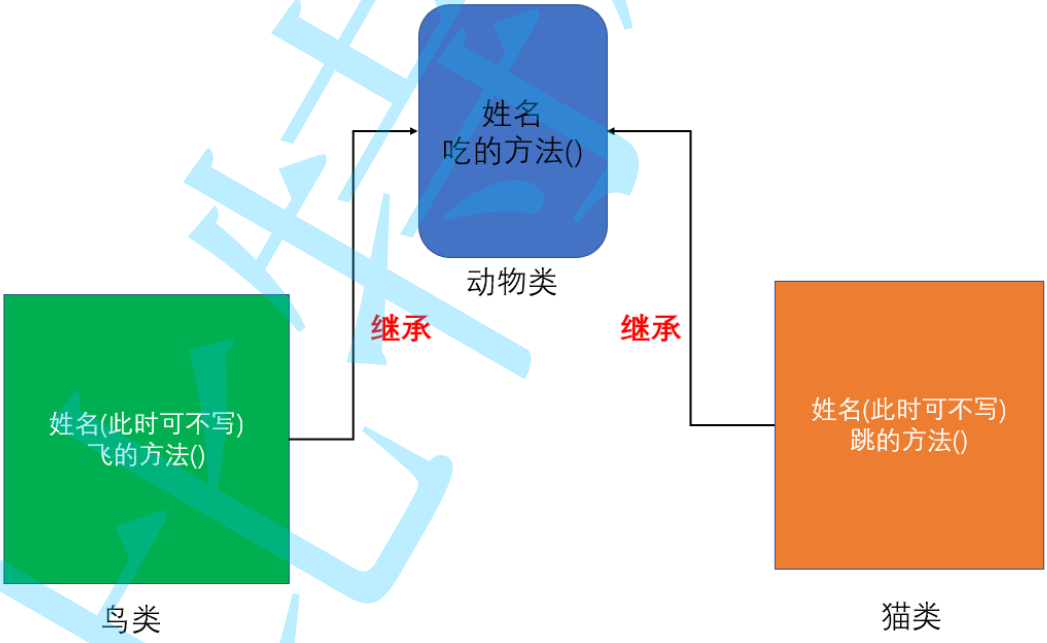
代码中创建的类, 主要是为了抽象现实中的一些事物(包含属性和方法).

有的时候客观事物之间就存在一些关联关系, 那么在表示成类和对象的时候也会存在一定的关联.

**继承(inheritance)机制**：是面向对象程序设计使代码可以复用的最重要的手段，它允许程序员在保持原有类特 性的基础上进行**扩展，增加功能**，这样产生新的类，称**派生类。**继承呈现了面向对象程序设计的层次结构， 体现了由简单到复杂的认知过程。

- 继承主要解决的问题是：共性的抽取

例如：鸟和猫都属于动物，那么我们就可以抽取出一些共性的内容。




- 鸟类如果继承了动物类，那么就不需要在鸟类当中再定义，姓名这个属性和吃的方法了。
- 也就是说子类可以拥有父类的内容
- 子类也可以拥有属于自己独有的内容

例如, 设计一个类表示动物

> 注意, 我们可以给每个类创建一个单独的 java 文件. 类名必须和 .java 文件名匹配(大小写敏感).

```
// Animal.java
```

```java
public class Animal {
    public String name;

    public Animal(String name) {
        this.name = name;
    }

    public void eat(String food) {
        System.out.println(this.name + "正在吃" + food);
    }
}

// Cat.java
class Cat {
    public String name;

    public Cat(String name) {
        this.name = name;
    }

    public void eat(String food) {
        System.out.println(this.name + "正在吃" + food);
    }
}

// Bird.java
class Bird {
    public String name;

    public Bird(String name) {
        this.name = name;
    }

    public void eat(String food) {
        System.out.println(this.name + "正在吃" + food);
    }

    public void fly() {
        System.out.println(this.name + "正在飞 ~(￣▽￣)~");
    }
}
```

这个代码我们发现其中存在了大量的冗余代码.

仔细分析, 我们发现 `Animal` 和 `Cat` 以及 `Bird` 这几个类中存在一定的关联关系:

- 这三个类都具备一个相同的 eat 方法, 而且行为是完全一样的.
- 这三个类都具备一个相同的 name 属性, 而且意义是完全一样的.
- 从逻辑上讲, Cat 和 Bird 都是一种 Animal (is - a 语义).

此时我们就可以让 Cat 和 Bird 分别继承 Animal 类, 来达到代码重用的效果.

此时, Animal 这样被继承的类, 我们称为 **父类** , **基类** 或 **超类**, 对于像 Cat 和 Bird 这样的类, 我们称为 **子类**, **派生类**

和现实中的儿子继承父亲的财产类似, 子类也会继承父类的字段和方法, 以达到代码重用的效果.

## 1.2 语法规则

**基本语法**

```
class 子类 extends 父类 {

}
```

- 使用 extends 指定父类.
- Java 中一个子类只能继承一个父类 (而C++/Python等语言支持多继承).
- 子类会继承父类的所有 public 的字段和方法.
- 对于父类的 private 的字段和方法, 子类中是无法访问的.
- 子类的实例中, 也包含着父类的实例. 可以使用 super 关键字得到父类实例的引用.

对于上面的代码, 可以使用继承进行改进. 此时我们让 Cat 和 Bird 继承自 Animal 类, 那么 Cat 在定义的时候就不必再写 name 字段和 eat 方法.

```
class Animal {
    public String name;

    public Animal(String name) {
        this.name = name;
    }

    public void eat(String food) {
        System.out.println(this.name + "正在吃" + food);
    }
}

class Cat extends Animal {
    public Cat(String name) {
        // 使用 super 调用父类的构造方法.
        super(name);
    }
}

class Bird extends Animal {
    public Bird(String name) {
        super(name);
    }

    public void fly() {
        System.out.println(this.name + "正在飞 ~(￣▽￣)~");
    }
}

public class Test {
    public static void main(String[] args) {
        Cat cat = new Cat("小黑");
        cat.eat("猫粮");
        Bird bird = new Bird("圆圆");
        bird.fly();
    }
}
```

> extends 英文原意指 "扩展". 而我们所写的类的继承, 也可以理解成基于父类进行代码上的 "扩展".

> 例如我们写的 Bird 类, 就是在 Animal 的基础上扩展出了 fly 方法.

**注意事项：**

- 当Cat类或者Bird类继承了Animal类之后，就会把父类的name继承过来,所以在子类当中，可以通过this关键字访问到name属性。
- 子类继承父类后，子类需要先构造父类。所以在子类的构造函数当中，要通过super()显示调用父类的构造方法。**super关键字的详细使用，会在本节课当中后面详细讲到**

**总结：**

> **继承的作用：**
>
> a. 代码复用的一种手段
>
> b. 用来实现多态（稍后会讲到）

如果我们把 name 改成 private, 那么此时子类就不能访问了.

```
class Bird extends Animal {
    public Bird(String name) {
        super(name);
    }

    public void fly() {
        System.out.println(this.name + "正在飞 ~(￣▽￣)~");
    }
}

// 编译出错
Error:(19, 32) java: name 在 Animal 中是 private 访问控制
```

## 1.3 对象的内存布局

当子类继承了父类之后，此时子类的内存布局应该是什么样子的？例如以下代码：

```
class Base {
    public int m;
}
class Derieve extends Base {
    public int n;
}
public class TestDemo {
    public static void main(String[] args) {
        Base base1 = new Base();//语句1
        Derieve derieve = new Derieve();//语句2
        Base base2 = new Derieve();//语句3
    }
}
```

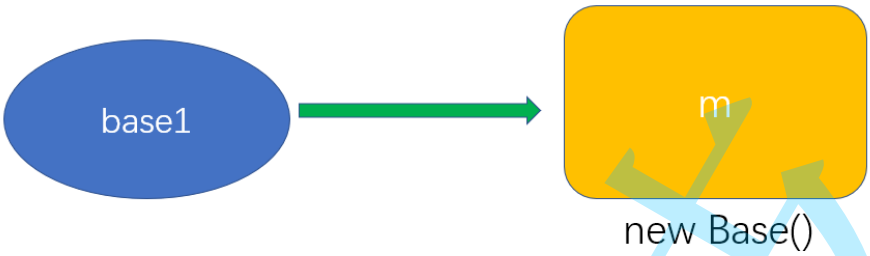
回顾引用：

> 引用：从本质上来讲，引用就是一个变量。比如：Base base1 = new Base();这句代码，base1就是一个引用变量，它指向了一个Base对象，也就是说：base1 引用了一个Base对象，我们通过操作base1来操作Base对象。base1 中存储的就是Base对象地址的哈希码。

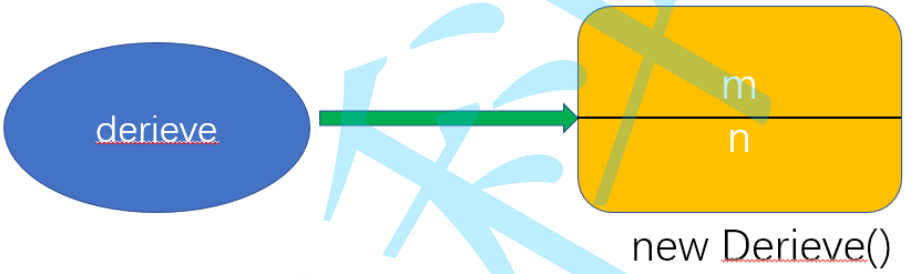
**注意事项：**

- base1，derieve，base2均为引用

语句1：

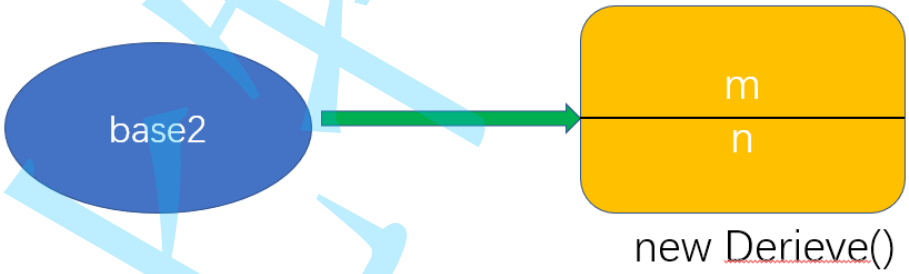



语句2：




子类继承了父类，在子类对象中，会存在父类的数据成员

语句3：

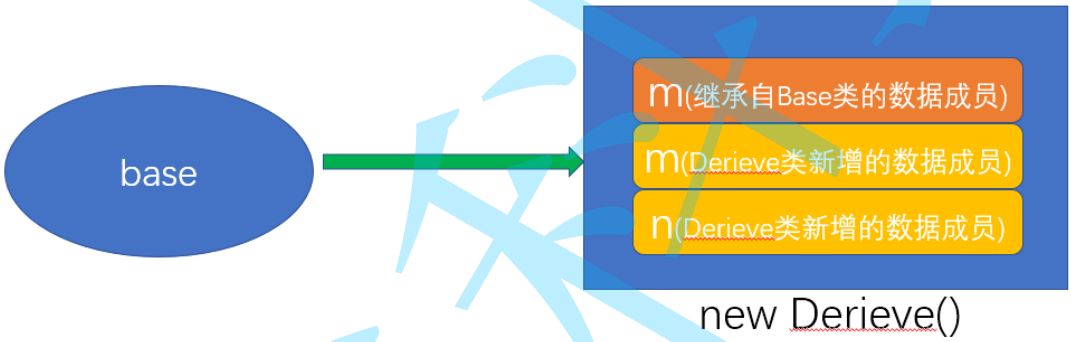



子类继承了父类，在子类对象中，会存在父类的数据成员

**当父类和子类都有同名的数据成员：**

```java
class Base {
    public int m;
}
class Derieve extends Base {
    public int m;
    public int n;
}
public class TestDemo {
    public static void main(String[] args) {
        Base base = new Derieve();
    }
}
```

待完成.....




子类继承了父类，在子类对象中，会存在父类的数据成员

**注意：子类当中如何访问父类当中的相同的数据成员?**

```java
class Base {
    public int m=10;
}
class Derieve extends Base {
    public int m=11;
    public int n;

    public void func() {
        System.out.println("访问Derieve类新增的数据成员m：+m);
        //本课件后面会详细讲到super关键字
        System.out.println("通过super访问父类的数据成员：+super.m);
    }
}

public class TestDemo {
    public static void main(String[] args) {
        Base base = new Derieve();
        System.out.println(base.m);
        Derieve derieve = new Derieve();
        System.out.println(derieve.m);
        System.out.println("=====子类当中如何访问，父类中的数据成员======");
        derieve.func();
    }
```

```
}
//执行结果
10
11
======子类当中如何访问，父类中的数据成员======
访问derieve类新增的数据成员m：11
通过super访问父类的数据成员：10
```

## 1.4 protected 关键字

刚才我们发现, 如果把字段设为 private, 子类不能访问. 但是设成 public, 又违背了我们 "封装" 的初衷.

两全其美的办法就是 protected 关键字.

- 对于类的调用者来说, protected 修饰的字段和方法是不能访问的
- 对于类的 **子类** 和 **同一个包的其他类** 来说, protected 修饰的字段和方法是可以访问的

```
// Animal.java
public class Animal {
    protected String name;

    public Animal(String name) {
        this.name = name;
    }

    public void eat(String food) {
        System.out.println(this.name + "正在吃" + food);
    }
}

// Bird.java
public class Bird extends Animal {
    public Bird(String name) {
        super(name);
    }

    public void fly() {
        // 对于父类的 protected 字段, 子类可以正确访问
        System.out.println(this.name + "正在飞 ~(￣▽￣)~");
    }
}

// Test.java 和 Animal.java 不在同一个 包 之中了.
public class Test {
    public static void main(String[] args) {
        Animal animal = new Animal("小动物");
        System.out.println(animal.name); // 此时编译出错, 无法访问 name
    }
}
```

**小结**: Java 中对于字段和方法共有四种访问权限

- private: 类内部能访问, 类外部不能访问
- 默认(也叫包访问权限): 类内部能访问, 同一个包中的类可以访问, 其他类不能访问.
- protected: 类内部能访问, 子类和同一个包中的类可以访问, 其他类不能访问.

- public : 类内部和类的调用者都能访问

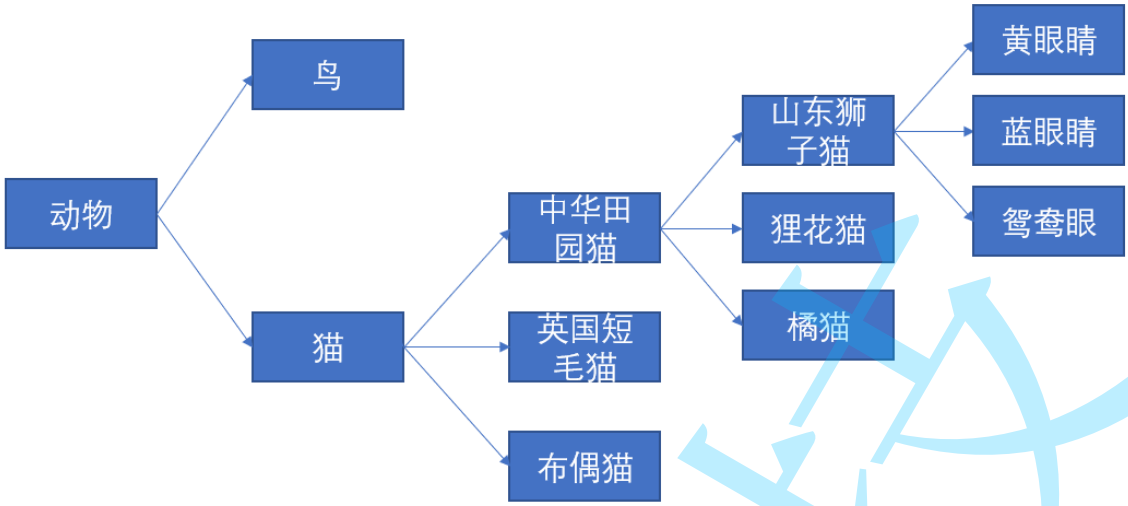
| No | 范围 | private | default | protected | public |
| --- | --- | --- | --- | --- | --- |
| 1 | 同一包中的同一类 | ✓ | ✓ | ✓ | ✓ |
| 2 | 同一包中的不同类 | | ✓ | ✓ | ✓ |
| 3 | 不同包中的子类 | | | ✓ | ✓ |
| 4 | 不同包中的非子类 | | | | ✓ |

> **什么时候下用哪一种呢?**
>
> 我们希望类要尽量做到 "封装", 即隐藏内部实现细节, 只暴露出 **必要** 的信息给类的调用者.
>
> 因此我们在使用的时候应该尽可能的使用 **比较严格** 的访问权限. 例如如果一个方法能用 private, 就尽量不要用 public.
>
> 另外, 还有一种 **简单粗暴** 的做法: 将所有的字段设为 private, 将所有的方法设为 public. 不过这种方式属于是对访问权限的滥用, 还是更希望同学们能写代码的时候认真思考, 该类提供的字段方法到底给 "谁" 使用(是类内部自己用, 还是类的调用者使用, 还是子类使用).

## 1.5 更复杂的继承关系

刚才我们的例子中, 只涉及到 Animal, Cat 和 Bird 三种类. 但是如果情况更复杂一些呢?

针对 Cat 这种情况, 我们可能还需要表示更多种类的猫~




这个时候使用继承方式来表示, 就会涉及到更复杂的体系.

```
// Animal.java
public Animal {
    ...
}

// Cat.java
public Cat extends Animal {
    ...
}

// ChineseGardenCat.java
public ChineseGardenCat extends Cat {
    ...
}

// OrangeCat.java
public Orange extends ChineseGardenCat {
    ...
}
......
```

如刚才这样的继承方式称为多层继承, 即子类还可以进一步的再派生出新的子类.

> 时刻牢记, 我们写的类是现实事物的抽象. 而我们真正在公司中所遇到的项目往往业务比较复杂, 可能会涉及到一系列复杂的概念, 都需要我们使用代码来表示, 所以我们真实项目中所写的类也会有很多. 类之间的关系也会更加复杂.
>
> 但是即使如此, 我们并不希望类之间的继承层次太复杂. 一般我们不希望出现超过三层的继承关系. 如果继承层次太多, 就需要考虑对代码进行重构了.
>
> 如果想从语法上进行限制继承, 就可以使用 final 关键字

## 1.6 final 关键字

曾经我们学习过 final 关键字, 修饰一个变量或者字段的时候, 表示 **常量** (不能修改).

```
final int a = 10;
a = 20;  // 编译出错
```

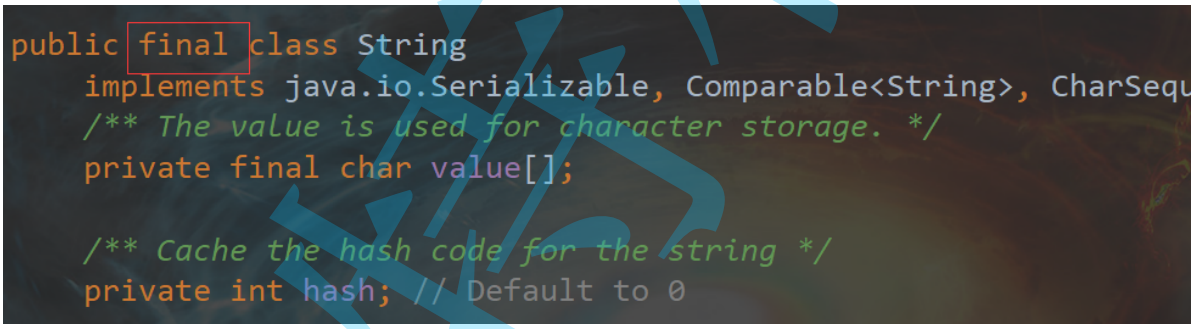
final 关键字也能修饰类, 此时表示被修饰的类就不能被继承.

```
final public class Animal {
    ...
}

public class Bird extends Animal {
    ...
}

// 编译出错
Error:(3, 27) java: 无法从最终com.bit.Animal进行继承
```

> final 关键字的功能是 **限制** 类被继承
>
> "限制" 这件事情意味着 "不灵活". 在编程中, 灵活往往不见得是一件好事. 灵活可能意味着更容易出错.
>
> 是用 final 修饰的类被继承的时候, 就会编译报错, 此时就可以提示我们这样的继承是有悖这个类设计的初衷的.




我们平时是用的 String 字符串类, 就是用 final 修饰的, 不能被继承.

# 2 组合

和继承类似, 组合也是一种表达类之间关系的方式, 也是能够达到代码重用的效果.

例如表示一个学校:

```
public class Student {
    ...
}

public class Teacher {
    ...
}

public class School {
    public Student[] students;
    public Teacher[] teachers;
}
```

组合并没有涉及到特殊的语法(诸如 extends 这样的关键字), 仅仅是将一个类的实例作为另外一个类的字段.

这是我们设计类的一种常用方式之一.

**组合表示 `has - a` 语义**

在刚才的例子中, 我们可以理解成一个学校中 "包含" 若干学生和教师.

**继承表示 `is - a` 语义**

在上面的 "动物和猫" 的例子中, 我们可以理解成一只猫也 "是" 一种动物.

大家要注意体会两种语义的区别.

# 3 多态

刚刚我们已经学过了继承了，继承其实就是实现多态的一个前提。在正式开始多态之前，我们首先来看看什么是向上转型。

## 3.1 向上转型

在刚才的例子中, 我们写了形如下面的代码

```
Bird bird = new Bird("圆圆");
```

这个代码也可以写成这个样子

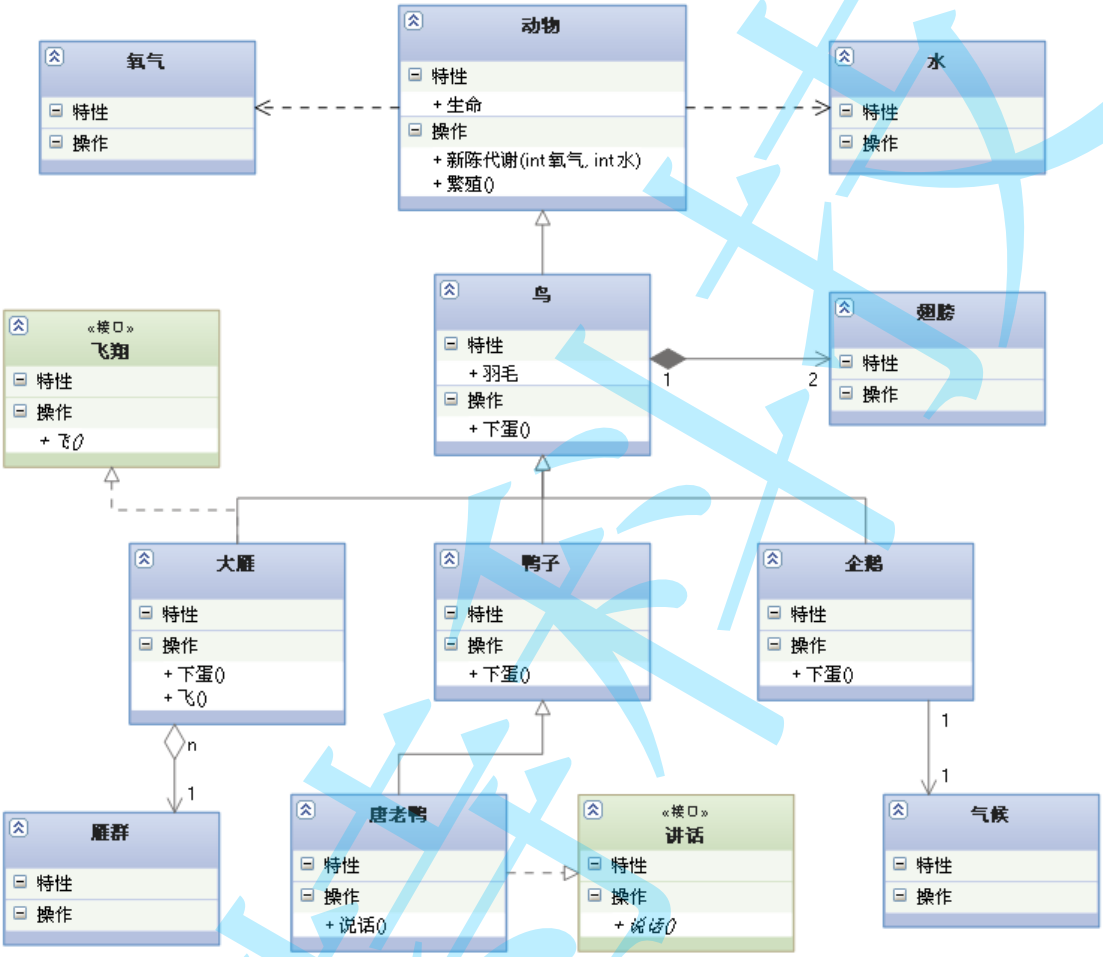
```
Bird bird = new Bird("圆圆");
Animal bird2 = bird;

// 或者写成下面的方式
Animal bird2 = new Bird("圆圆");
```

此时 bird2 是一个父类 (Animal) 的引用, 指向一个子类 (Bird) 的实例. 这种写法称为 **向上转型**.

> 向上转型这样的写法可以结合 `is - a` 语义来理解.
>
> 例如, 我让我媳妇去喂圆圆, 我就可以说, "媳妇你喂小鸟了没?", 或者 "媳妇你喂鹦鹉了没?"
>
> 因为圆圆确实是一只鹦鹉, 也确实是一只小鸟~~

> **为啥叫 "向上转型"?**
>
> 在面向对象程序设计中, 针对一些复杂的场景(很多类, 很复杂的继承关系), 程序猿会画一种 UML 图的方式来表示类之间的关系. 此时父类通常画在子类的上方. 所以我们就称为 "向上转型" , 表示往父类的方向转.
>
> 注意: 关于 UML 图的规则我们课堂上不详细讨论, 有兴趣的同学自己看看即可.




向上转型发生的时机:

- 直接赋值
- 方法传参
- 方法返回

直接赋值的方式我们已经演示了. 另外两种方式和直接赋值没有本质区别.

**方法传参**

```java
public class Test {
    public static void main(String[] args) {
        Bird bird = new Bird("圆圆");
        feed(bird);
    }

    public static void feed(Animal animal) {
        animal.eat("谷子");
    }
}

// 执行结果
圆圆正在吃谷子
```

此时形参 animal 的类型是 Animal (基类), 实际上对应到 Bird (父类) 的实例.

**方法返回**

```java
public class Test {
    public static void main(String[] args) {
        Animal animal = findMyAnimal();
    }

    public static Animal findMyAnimal() {
        Bird bird = new Bird("圆圆");
        return bird;
    }
}
```

此时方法 `findMyAnimal` 返回的是一个 Animal 类型的引用, 但是实际上对应到 Bird 的实例.

## 3.2 动态绑定

当子类和父类中出现同名方法的时候, 再去调用会出现什么情况呢?

对前面的代码稍加修改, 给 Bird 类也加上同名的 eat 方法, 并且在两个 eat 中分别加上不同的日志.

```java
// Animal.java
public class Animal {
    protected String name;

    public Animal(String name) {
        this.name = name;
    }

    public void eat(String food) {
        System.out.println("我是一只小动物");
        System.out.println(this.name + "正在吃" + food);
    }
}

// Bird.java
public class Bird extends Animal {
    public Bird(String name) {
        super(name);
```

```
    }

    public void eat(String food) {
        System.out.println("我是一只小鸟");
        System.out.println(this.name + "正在吃" + food);
    }
}

// Test.java
public class Test {
    public static void main(String[] args) {
        Animal animal1 = new Animal("圆圆");
        animal1.eat("谷子");
        Animal animal2 = new Bird("扁扁");
        animal2.eat("谷子");
    }
}

// 执行结果
我是一只小动物
圆圆正在吃谷子
我是一只小鸟
扁扁正在吃谷子
```

此时, 我们发现:

- animal1 和 animal2 虽然都是 `Animal` 类型的引用, 但是 animal1 指向 `Animal` 类型的实例, animal2 指向 `Bird` 类型的实例.
- 针对 animal1 和 animal2 分别调用 eat 方法, 发现 `animal1.eat()` 实际调用了父类的方法, 而 `animal2.eat()` 实际调用了子类的方法.

因此, 在 Java 中, 调用某个类的方法, 究竟执行了哪段代码 (是父类方法的代码还是子类方法的代码) , 要看究竟这个引用指向的是父类对象还是子类对象. 这个过程是程序运行时决定的(而不是编译期), 因此称为**动态绑定**.

## 3.3 方法重写

针对刚才的 eat 方法来说:

子类实现父类的同名方法, 并且参数的类型和个数完全相同, 这种情况称为 **覆写/重写/覆盖(Override)**.

**关于重写的注意事项**

1. 重写和重载完全不一样. 不要混淆(思考一下, 重载的规则是啥?)
2. 普通方法可以重写, static 修饰的静态方法不能重写.
3. 重写中子类的方法的访问权限不能低于父类的方法访问权限.
4. 重写的方法返回值类型不一定和父类的方法相同(但是建议最好写成相同, 特殊情况除外).

**方法权限示例**: 将子类的 eat 改成 private

```
// Animal.java
public class Animal {
    public void eat(String food) {
        ...
    }
}

// Bird.java
```

```
public class Bird extends Animal {
    // 将子类的 eat 改成 private
    private void eat(String food) {
        ...
    }
}

// 编译出错
Error:(8, 10) java: com.bit.Bird中的eat(java.lang.String)无法覆盖com.bit.Animal中的
eat(java.lang.String)
  正在尝试分配更低的访问权限; 以前为public
```

另外, 针对重写的方法, 可以使用 `@Override` 注解来显式指定.

```
// Bird.java
public class Bird extends Animal {
    @Override
    private void eat(String food) {
        ...
    }
}
```

有了这个注解能帮我们进行一些合法性校验. 例如不小心将方法名字拼写错了 (比如写成 aet), 那么此时编译器就会发现父类中没有 aet 方法, 就会编译报错, 提示无法构成重写.

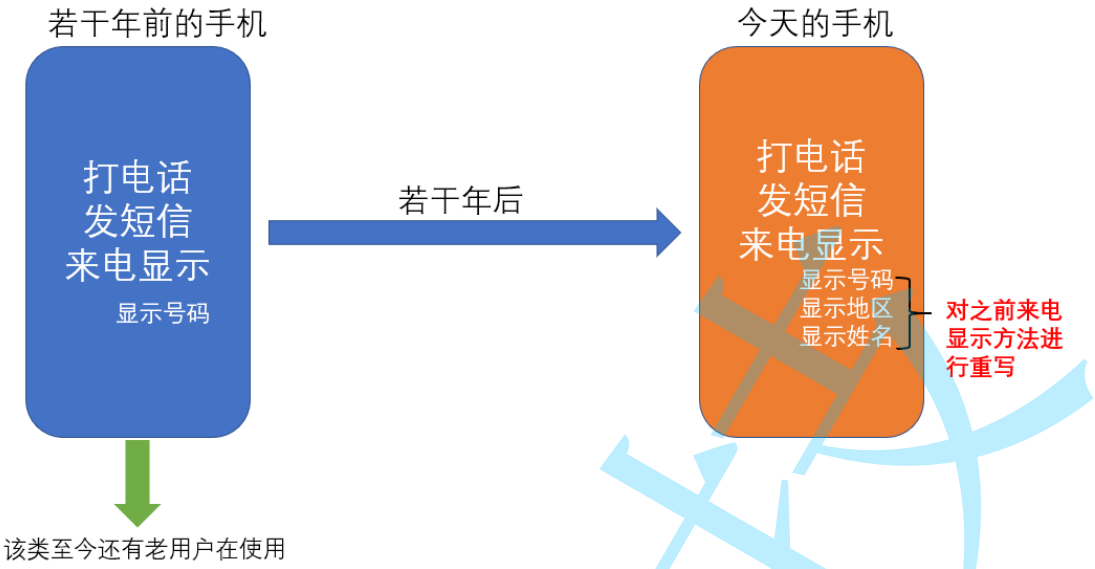
我们推荐在代码中进行重写方法时**显式加上** @Override 注解.

> 关于注解的详细内容, 我们会在后面的章节再详细介绍.

## 3.4 重写的设计原则

对于已经投入使用的类，尽量不要进行修改。最好的方式是：重新定义一个新的类，来重复利用其中共性的内容，并且添加或者改动新的内容。

例如：若干年前的手机，只能打电话，发短信，来电显示只能显示号码，而今天的手在来电显示的时候，不仅仅可以显示号码，还可以显示头像，地区等。在这个过程当中，我们**不应该在原来的老的类上进行修改，因为原来的类，可能还在有用户使用**，我们的正确做法是：**新建一个新手机的类，对来电显示这个方法重写就好了，这样就达到了我们当今的需求了。**




## 3.5 重载和重写的区别

| No | 区别 | 重载（overload） | 覆写(override) |
| --- | --- | --- | --- |
| 1 | 概念 | 方法名称相同，参数的类型及个数不同 | 方法名称、返回值类型、参数的类型及个数完全相同 |
| 2 | 范围 | 一个类 | 继承关系 |
| 3 | 限制 | 没有权限要求 | 被覆写的方法不能拥有比父类更严格的访问控制权限 |

> **体会动态绑定和方法重写**
>
> 上面讲的动态绑定和方法重写是用的相同的代码示例.
>
> 事实上, 方法重写是 Java 语法层次上的规则, 而动态绑定是方法重写这个语法规则的底层实现. 两者本质上描述的是相同的事情, 只是侧重点不同.

## 3.6 理解多态

有了面的向上转型, 动态绑定, 方法重写之后, 我们就可以使用 **多态(polypeptide)** 的形式来设计程序了.

我们可以写一些只关注父类的代码, 就能够同时兼容各种子类的情况.

**代码示例**: 打印多种形状

```
class Shape {
    public void draw() {
        // 啥都不用干
    }
}

class Cycle extends Shape {
    @Override
    public void draw() {
        System.out.println("o");
    }
}

class Rect extends Shape {
    @Override
    public void draw() {
        System.out.println("□");
    }
}

class Flower extends Shape {
    @Override
    public void draw() {
        System.out.println("❀");
    }
}

//////////////////////////我是分割线/////////////////////

// Test.java
public class Test {
    public static void main(String[] args) {
        Shape shape1 = new Flower();
        Shape shape2 = new Cycle();
        Shape shape3 = new Rect();
        drawMap(shape1);
        drawMap(shape2);
        drawMap(shape3);
    }
    // 打印单个图形
    public static void drawShape(Shape shape) {
        shape.draw();
    }
}
```

在这个代码中, 分割线上方的代码是 **类的实现者** 编写的, 分割线下方的代码是 **类的调用者** 编写的.

当类的调用者在编写 `drawMap` 这个方法的时候, 参数类型为 Shape (父类), 此时在该方法内部并**不知道, 也不关注**当前的 shape 引用指向的是哪个类型(哪个子类)的实例. 此时 shape 这个引用调用 draw 方法可能会有多种不同的表现(和 shape 对应的实例相关), 这种行为就称为 **多态**.

> 多态顾名思义, 就是 "一个引用, 能表现出多种不同形态"
>
> 举个具体的例子. 汤老湿家里养了两只鹦鹉(圆圆和扁扁)和一个小孩(核弹). 我媳妇管他们都叫 "儿子". 这时候我对我媳妇说, "你去喂喂你儿子去". 那么如果这里的 "儿子" 指的是鹦鹉, 我媳妇就要喂鸟粮; 如果这里的 "儿子" 指的是核弹, 我媳妇就要喂馒头.
>
> 那么如何确定这里的 "儿子" 具体指的是啥? 那就是根据我和媳妇对话之间的 "上下文".
>
> 代码中的多态也是如此. 一个引用到底是指向父类对象, 还是某个子类对象(可能有多个), 也是要根据上下文的代码来确定.
>
> PS: 大家可以根据汤老湿说话的语气推测一下在家里的家庭地位.

**使用多态的好处是什么?**

**1) 类调用者对类的使用成本进一步降低.**

- 封装是让类的调用者不需要知道类的实现细节.
- 多态能让类的调用者连这个类的类型是什么都不必知道, 只需要知道这个对象具有某个方法即可.

因此, 多态可以理解成是封装的更进一步, 让类调用者对类的使用成本进一步降低.

这也贴合了 <<代码大全>> 中关于 "管理代码复杂程度" 的初衷.

**2) 能够降低代码的 "圈复杂度", 避免使用大量的 if - else**

例如我们现在需要打印的不是一个形状了, 而是多个形状. 如果不基于多态, 实现代码如下:

```
public static void drawShapes() {
    Rect rect = new Rect();
    Cycle cycle = new Cycle();
    Flower flower = new Flower();
    String[] shapes = {"cycle", "rect", "cycle", "rect", "flower"};

    for (String shape : shapes) {
        if (shape.equals("cycle")) {
            cycle.draw();
        } else if (shape.equals("rect")) {
            rect.draw();
        } else if (shape.equals("flower")) {
            flower.draw();
        }
    }
}
```

如果使用使用多态, 则不必写这么多的 if - else 分支语句, 代码更简单.

```
public static void drawShapes() {
    // 我们创建了一个 Shape 对象的数组.
    Shape[] shapes = {new Cycle(), new Rect(), new Cycle(),
                      new Rect(), new Flower()};
    for (Shape shape : shapes) {
        shape.draw();
    }
}
```

> 什么叫 "圈复杂度" ?
>
> 圈复杂度是一种描述一段代码复杂程度的方式. 一段代码如果平铺直叙, 那么就比较简单容易理解. 而如果有很多的条件分支或者循环语句, 就认为理解起来更复杂.
>
> 因此我们可以简单粗暴的计算一段代码中条件语句和循环语句出现的个数, 这个个数就称为 "圈复杂度". 如果一个方法的圈复杂度太高, 就需要考虑重构.
>
> 不同公司对于代码的圈复杂度的规范不一样. 一般不会超过 10 .

**3) 可扩展能力更强.**

如果要新增一种新的形状, 使用多态的方式代码改动成本也比较低.

```
class Triangle extends Shape {
    @Override
    public void draw() {
        System.out.println("△");
    }
}
```

对于类的调用者来说(drawShapes方法), 只要创建一个新类的实例就可以了, 改动成本很低.

而对于不用多态的情况, 就要把 drawShapes 中的 if - else 进行一定的修改, 改动成本更高.

## 3.7 向下转型

向上转型是子类对象转成父类对象, 向下转型就是父类对象转成子类对象. 相比于向上转型来说, 向下转型没那么常见, 但是也有一定的用途.

```
// Animal.java
public class Animal {
    protected String name;

    public Animal(String name) {
        this.name = name;
    }

    public void eat(String food) {
        System.out.println("我是一只小动物");
        System.out.println(this.name + "正在吃" + food);
    }
}

// Bird.java
```

```java
public class Bird extends Animal {
    public Bird(String name) {
        super(name);
    }

    public void eat(String food) {
        System.out.println("我是一只小鸟");
        System.out.println(this.name + "正在吃" + food);
    }

    public void fly() {
        System.out.println(this.name + "正在飞");
    }
}
```

接下来是我们熟悉的操作

```java
Animal animal = new Bird("圆圆");
animal.eat("谷子");

// 执行结果
圆圆正在吃谷子
```

接下来我们尝试让圆圆飞起来

```java
animal.fly();

// 编译出错
找不到 fly 方法
```

**注意事项**

编译过程中, animal 的类型是 Animal, 此时编译器只知道这个类中有一个 eat 方法, 没有 fly 方法.

虽然 animal 实际引用的是一个 Bird 对象, 但是编译器是以 animal 的类型来查看有哪些方法的.

对于 `Animal animal = new Bird("圆圆")` 这样的代码,

- 编译器检查有哪些方法存在, 看的是 Animal 这个类型
- 执行时究竟执行父类的方法还是子类的方法, 看的是 Bird 这个类型.

那么想实现刚才的效果, 就需要向下转型.

```java
// (Bird) 表示强制类型转换
Bird bird = (Bird)animal;
bird.fly();

// 执行结果
圆圆正在飞
```

但是这样的向下转型有时是不太可靠的. 例如

```
Animal animal = new Cat("小猫");
Bird bird = (Bird)animal;
bird.fly();

// 执行结果, 抛出异常
Exception in thread "main" java.lang.ClassCastException: Cat cannot be cast to
Bird
    at Test.main(Test.java:35)
```

animal 本质上引用的是一个 Cat 对象, 是不能转成 Bird 对象的. 运行时就会抛出异常.

所以, 为了让向下转型更安全, 我们可以先判定一下看看 animal 本质上是不是一个 Bird 实例, 再来转换

```
Animal animal = new Cat("小猫");
if (animal instanceof Bird) {
    Bird bird = (Bird)animal;
    bird.fly();
}
```

`instanceof` 可以判定一个引用是否是某个类的实例. 如果是, 则返回 true. 这时再进行向下转型就比较安全了.

## 3.8 super 关键字

前面的代码中由于使用了重写机制, 调用到的是子类的方法. 如果需要在子类内部调用父类方法怎么办? 可以使用 super 关键字.

**super 表示获取到父类实例的引用**. 涉及到两种常见用法.

1) 使用了 super 来调用父类的构造器(这个代码前面已经写过了)

```
public Bird(String name) {
    super(name);
}
```

2) 使用 super 来调用父类的普通方法

```
public class Bird extends Animal {
    public Bird(String name) {
        super(name);
    }

    @Override
    public void eat(String food) {
        // 修改代码, 让子调用父类的接口.
        super.eat(food);
        System.out.println("我是一只小鸟");
        System.out.println(this.name + "正在吃" + food);
    }
}
```

在这个代码中, 如果在子类的 eat 方法中直接调用 eat (不加super), 那么此时就认为是调用子类自己的 eat (也就是递归了). 而加上 super 关键字, 才是调用父类的方法.

注意 super 和 this 功能有些相似, 但是还是要注意其中的区别.

| No | 区别 | this | super |
| --- | --- | --- | --- |
| 1 | 概念 | 访问本类中的属性和方法 | 由子类访问父类中的属性、方法 |
| 2 | 查找范围 | 先查找本类，如果本类没有就调用父类 | 不查找本类而直接调用不累定义 |
| 3 | 特殊 | 表示当前对象 | 无 |

## 3.9 在构造方法中调用重写的方法(一个坑)

一段有坑的代码. 我们创建两个类, B 是父类, D 是子类. D 中重写 func 方法. 并且在 B 的构造方法中调用 func

```
class B {
    public B() {
        // do nothing
        func();
    }

    public void func() {
        System.out.println("B.func()");
    }
}

class D extends B {
    private int num = 1;
    @Override
    public void func() {
        System.out.println("D.func() " + num);
    }
}

public class Test {
    public static void main(String[] args) {
        D d = new D();
    }
}
```

```
// 执行结果
D.func() 0
```

- 构造 D 对象的同时, 会调用 B 的构造方法.
- B 的构造方法中调用了 func 方法, 此时会触发动态绑定, 会调用到 D 中的 func
- 此时 D 对象自身还没有构造, 此时 num 处在未初始化的状态, 值为 0.

**结论:** "用尽量简单的方式使对象进入可工作状态", 尽量不要在构造器中调用方法(如果这个方法被子类重写, 就会触发动态绑定, 但是此时子类对象还没构造完成), 可能会出现一些隐藏的但是又极难发现的问题.

### 3.10 总结

多态是面向对象程序设计中比较难理解的部分. 我们会在后面的抽象类和接口中进一步体会多态的使用. 重点是多态带来的编码上的好处.

另一方面, 如果抛开 Java, 多态其实是一个更广泛的概念, 和 "继承" 这样的语法并没有必然的联系.

- C++ 中的 "动态多态" 和 Java 的多态类似. 但是 C++ 还有一种 "静态多态"(模板), 就和继承体系没有关系了.
- Python 中的多态体现的是 "鸭子类型", 也和继承体系没有关系.
- Go 语言中没有 "继承" 这样的概念, 同样也能表示多态.

无论是哪种编程语言, 多态的核心都是让调用者**不必关注对象的具体类型**. 这是降低用户使用成本的一种重要方式.

## 4 执行顺序

我们还记得之前讲过的代码块吗？我们简单回顾一下几个重要的代码块：实例代码块和静态代码块。在没有继承关系时的执行顺序。

```
class Person {
    public String name;
    public int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
        System.out.println("构造方法执行");
    }
    {
        System.out.println("实例代码块执行");
    }
    static {
        System.out.println("静态代码块执行");
    }
}

public class TestDemo {
    public static void main(String[] args) {
        Person person1 = new Person("bit",10);
        System.out.println("============================");
        Person person2 = new Person("gaobo",20);
    }
```

```
}
```

执行结果：

```
静态代码块执行
实例代码块执行
构造方法执行
============================
实例代码块执行
构造方法执行
```

> 1、静态代码块只执行一次
>
> 2、静态代码块先执行，实例代码块接着执行，最后构造方法执行

## 继承关系上的执行顺序

```
class Person {
    public String name;
    public int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
        System.out.println("Person：构造方法执行");
    }
    {
        System.out.println("Person：实例代码块执行");
    }
    static {
        System.out.println("Person：静态代码块执行");
    }
}

class Student extends Person{

    public Student(String name,int age) {
        super(name,age);
        System.out.println("Student：构造方法执行");
    }

    {
        System.out.println("Student：实例代码块执行");
    }
    static {
        System.out.println("Student：静态代码块执行");
    }


}

public class TestDemo4 {

    public static void main(String[] args) {
        Student student1 = new Student("张三",19);
        System.out.println("============================");
        Student student2 = new Student("gaobo",20);
```

```
    }

    public static void main1(String[] args) {
        Person person1 = new Person("bit",10);
        System.out.println("============================");
        Person person2 = new Person("gaobo",20);
    }
}
```

执行结果：

```
Person：静态代码块执行
Student：静态代码块执行
Person：实例代码块执行
Person：构造方法执行
Student：实例代码块执行
Student：构造方法执行
============================
Person：实例代码块执行
Person：构造方法执行
Student：实例代码块执行
Student：构造方法执行
```

> 通过分析执行结果，得出以下结论：
>
> 1、父类静态代码块优先于子类静态代码块执行，且是最早执行。
>
> 2、父类实例代码块和父类构造方法紧接着执行
>
> 3、子类的实例代码块和子类构造方法紧接着再执行
>
> 4、第二次实例化子类对象时，父类和子类的静态代码块都将不会再执行。