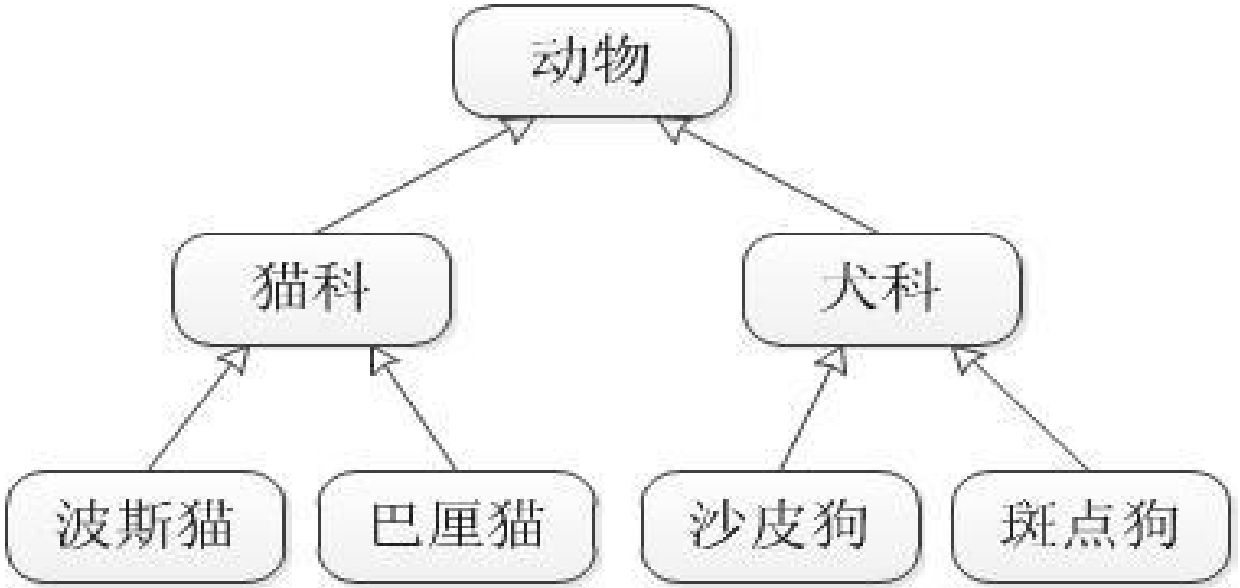
第4章 面向对象（下）

- 类的继承
- final关键字
- 抽象类与接口
- 多态
- 内部类
- 异常

4.1.1 继承的概念

在现实生活中，继承一般指的是子女继承父辈的财产。在程序中，继承描述的是事物之间的所属关系，通过继承可以使多种事物之间形成一种关系体系。例如，猫和狗都属于动物，程序中便可以描述为猫和狗继承自动物，同理，波斯猫和巴厘猫继承猫科，而沙皮狗和斑点狗继承自犬科。

4.1.1 继承的概念



4.1.1 继承的概念

在Java中，类的继承是指在一个现有类的基础上去构建一个新的类，构建出来的新类被称作子类，现有类被称作父类。子类继承父类的属性和方法，使得子类对象（实例）具有父类的特征和行为。

4.1.1 继承的概念

在程序中，如果想声明一个类继承另一个类，需要使用extends关键字，语法格式如下所示。

```
class 父类{
  ……
 }
class 子类 extends 父类{
  ……
}
```

4.1.1 继承的概念

接下来通过一个案例学习子类是如何继承父类。

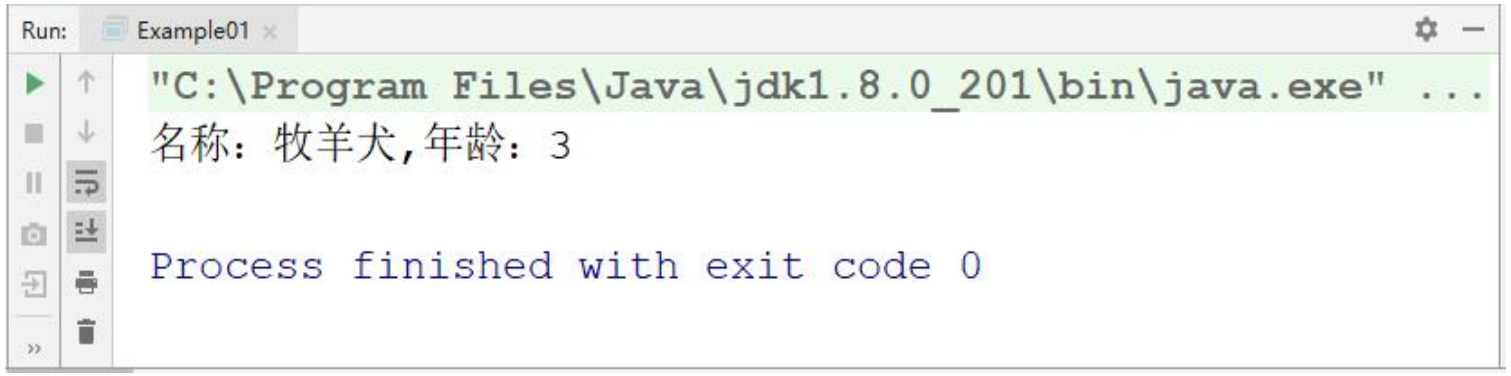
```
1 // 定义Animal类
2 class Animal {
3    private String name;   // 定义name属性
4    private int age;          // 定义name属性
5    public String getName() {
6        return name;
7    }
8    public void setName(String name) {
9        this.name = name;
10   }
11 public int getAge() {
12       return age;
13   }
14 public void setAge(int  age) {
15       this.age = age;
16   }
17 }
18 // 定义Dog类继承Animal类
19 class Dog extends Animal {
20   //此处不写任何代码
21 }
```

4.1.1 继承的概念

```
22 // 定义测试类
23 public class Example01 {
24    public static void main(String[] args) {
25   Dog dog = new Dog();                    // 创建一个Dog类的实例对象
26       dog.setName("牧羊犬");               // 此时访问的方法时父类中的，子类中并没有定义
27 dog.setAge(3);                            // 此时访问的方法时父类中的，子类中并没有定义
28 System.out.println("名称："+dog.getName()+",年龄："+dog.getAge());
29    }
30 }
```

4.1.1 继承的概念

程序运行结果如下图。



4.1.1 继承的概念

上述代码中，第2~17行代码定义了一个Animal类，第19~21行代码定义了一个Dog类，Dog类中并没有定义任何操作，而是通过extends关键字继承了Animal类，成为了Animal类的子类。从运行结果可以看出，子类虽然没有定义任何属性和方法，但是却能调用父类的方法。这就说明，子类在继承父类的时候，会自动继承父类的成员。

4.1.1 继承的概念

除了继承父类的属性和方法，子类也可以定义自己的属性和方法。

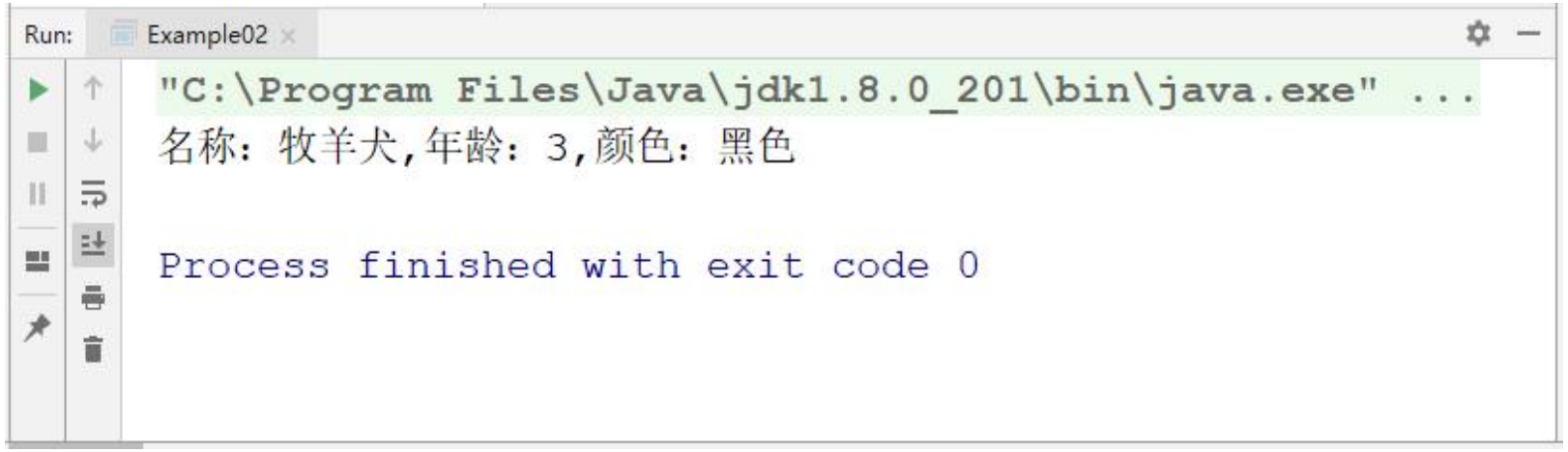
```
1 // 定义Animal类
2 class Animal {
3    private String name; // 定义name属性
4    private int age;        // 定义name属性
5    public String getName() {
6        return name;
7    }
8    public void setName(String name) {
9        this.name = name;
10   }
11 public int getAge() {
12       return age;
13   }
14 public void setAge(int age) {
15       this.age = age;
16   }
17 }
18 // 定义Dog类继承Animal类
19class Dog extends Animal {
20   private String color; // 定义name属性
21   public String getColor() {
22   return color;
23 }
24 public void setColor(String color) {
25   this.color = color;
26 }
27 }
```

4.1.1 继承的概念

```
28 // 定义测试类
29 public class Example02 {
30     public static void main(String[] args) {
31         Dog dog = new Dog();    // 创建一个Dog类的实例对象
32         dog.setName("牧羊犬");   // 此时访问的方法时父类中的，子类中并没有定义
33         dog.setAge(3);          // 此时访问的方法时父类中的，子类中并没有定义
34         dog.setColor("黑色");
35         System.out.println("名称："+dog.getName()+",年龄："+dog.getAge()+",
36         颜色："+dog.getColor());
37     }
38 }
```

4.1.1 继承的概念

程序运行结果如下图。



4.1.1 继承的概念

在上述代码中，Dog类扩充了Animal类，增加了color属性及getColor()和setColor()方法。此时的Dog类已经存在了3个属性和6个方法。在main()方法中，第31行代码创建并实例化dog对象；第32~34行代码通过dog对象调用Animal类和Dog类的setter方法，设置名称、年龄和颜色；第35行代码通过dog对象调用Animal类和Dog类的getter方法获取名称、年龄和颜色。由运行结果可知，程序成功设置并获取了dog对象的名称、年龄和和颜色。

4.1.1 继承的概念

继承中需要注意的问题。

（1）在Java中，类只支持单继承，不允许多重继承。也就是说一个类只能有一个直接父类，例如下面这种情况是不合法的。

```
class A{}
class B{}
class C extends A,B{}  // C类不可以同时继承A类和B类
```

4.1.1 继承的概念

（2）多个类可以继承一个父类，例如下面这种情况是允许的。

```
class A{}
class B extends A{}
class C extends A{}   // 类B和类C都可以继承类A
```

4.1.1 继承的概念

（3）在Java中，多层继承也是可以的，即一个类的父类可以再继承另外的父类。例如，C类继承自B类，而B类又可以继承自A类，这时，C类也可称作A类的子类。例如下面这种情况是允许的。

```
class A{}
class B extends A{}   // 类B继承类A，类B是类A的子类
class C extends B{}   // 类C继承类B，类C是类B的子类，同时也是类A的子类
```

4.1.1 继承的概念

（4）在Java中，子类和父类是一种相对概念，一个类可以是某个类的父类，也可以是另一个类的子类。例如，在第（3）种情况中，B类是A类的子类，同时又是C类的父类。

在继承中，子类不能直接访问父类中的私有成员，子类可以调用父类的非私有方法，但是不能调用父类的私有成员。

4.1.2 方法的重写

在继承关系中，子类会自动继承父类中定义的方法，但有时在子类中需要对继承的方法进行一些修改，即对父类的方法进行重写。在子类中重写的方法需要和父类被重写的方法具有相同的方法名、参数列表以及返回值类型，且在子类重写的方法不能拥有比父类方法更加严格的访问权限。

4.1.2 方法的重写

下面通过一个案例讲解方法的重写。

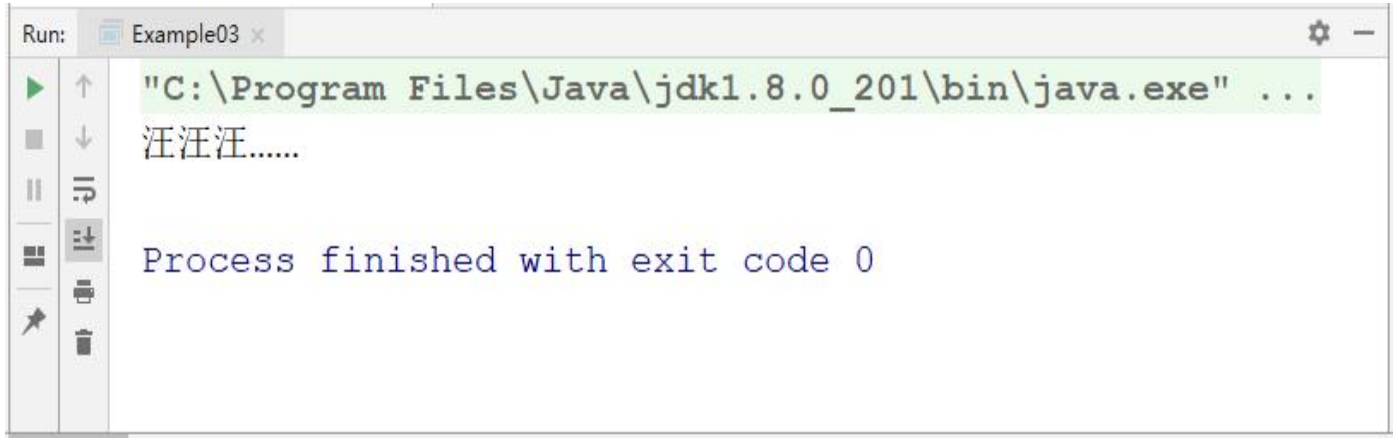
```
1 // 定义Animal类
2 class Animal {
3    //定义动物叫的方法
4            void shout() {
5                        System.out.println("动物发出叫声");
6            }
7 }
8 // 定义Dog类继承动物类
9 class Dog extends Animal {
10           //重写父类Animal中的shout()方法
11           void shout() {
12                       System.out.println("汪汪汪……");
13           }
14 }
```

4.1.2 方法的重写

```
15 // 定义测试类
16 public class Example03 {
17         public static void main(String[] args) {
18                 Dog dog = new Dog(); // 创建Dog类的实例对象
19                 dog.shout();          // 调用dog重写的shout()方法
20         }
21 }
```

4.1.2 方法的重写

程序运行结果如下图。



4.1.2 方法的重写

上述代码中，第2~7行代码定义了一个Animal类，并在Animal类中定义了一个shout()方法。第9~14行代码定义了Dog类继承Animal类，并在类中重写了父类Aniaml的shout()方法。第18~19行代码创建并实例化Dog类对象dog，并通过dog对象调用shout()方法。从图4-4可以看出，dog对象调用的是子类重写的shout()方法，而不是父类的shout()方法。

脚下留心：子类重写父类方法时的访问权限

子类重写父类方法时，不能使用比父类中被重写的方法更严格的访问权限。例如，父类中的方法是public权限，子类的方法就不能是private权限。如果子类在重写父类方法时定义的权限缩小，则在编译时将出现错误提示。

脚下留心：子类重写父类方法时的访问权限

```
1 // 定义Animal类
2 class Animal {
3     //定义动物叫的方法
4   public void shout() {
5                 System.out.println("动物发出叫声");
6         }
7 }
8 // 定义Dog类继承动物类
9 class Dog extends Animal {
10        //重写父类Animal中的shout()方法
11        private void shout() {
12                System.out.println("汪汪汪……");
13        }
14 }
```

脚下留心：子类重写父类方法时的访问权限

```
15 // 定义测试类
16 public class Example04 {
17      public static void main(String[] args) {
18              Dog dog = new Dog(); // 创建Dog类的实例对象
19              dog.shout();         // 调用dog重写的shout()方法
20      }
21 }
```

脚下留心：子类重写父类方法时的访问权限

程序运行结果如下图。

脚下留心：子类重写父类方法时的访问权限

在上述代码中，第4行代码在Animal类中定义了一个shout()方法并将访问权限定义为public，第9~14行代码定义了一个Dog类并继承Animal类，第11行代码在声明shout()方法时，将shou()方法的访问权限定义为private。由运行结果可知，编译文件会报错，这是因为子类重写父类方法时，不能使用比父类中被重写的方法更严格的访问权限。

4.1.3 super关键字

当子类重写父类的方法后，子类对象将无法访问父类被重写的方法，为了解决这个问题，Java提供了super关键字，super关键字可以在子类中调用父类的普通属性、方法以及构造方法。

4.1.3 super关键字

接下来详细讲解super关键字的具体用法。

（1）使用super关键字访问父类的成员变量和成员方法，具体格式如下：

super.成员变量

super.成员方法(参数1,参数2...)

4.1.3 super关键字

接下来通过一个案例学习super关键字访问父类的成员变量和成员方法。

```
1 // 定义Animal类
2 class Animal {
3    String name = "牧羊犬";
4    //定义动物叫的方法
5          void shout() {
6                    System.out.println("动物发出叫声");
7          }
8 }
```

4.1.3 super关键字

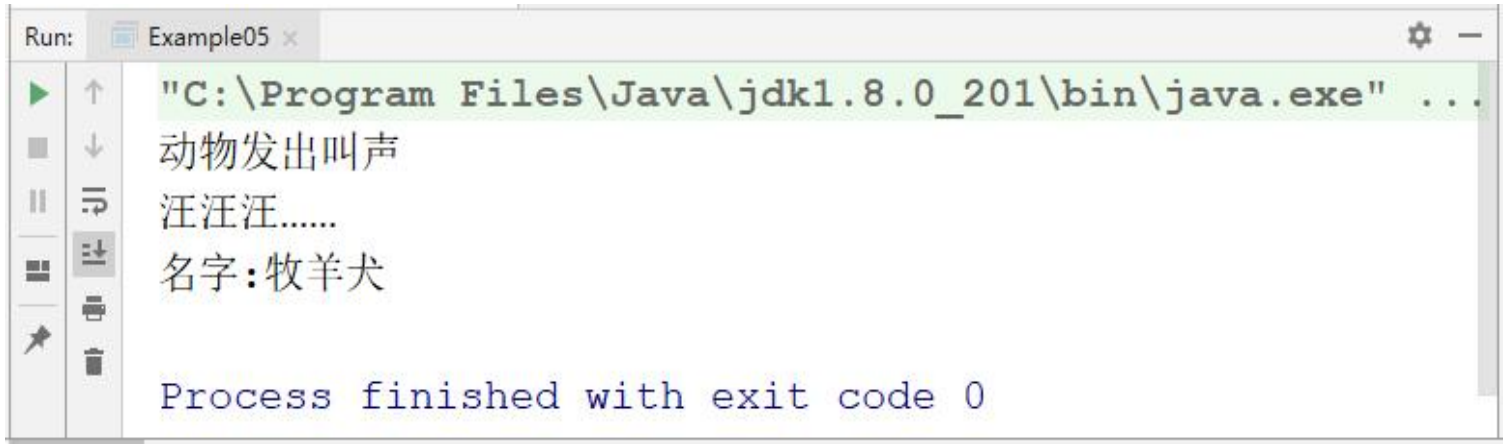
```
9 // 定义Dog类继承动物类
10 class Dog extends Animal {
11     //重写父类Animal中的shout()方法,扩大了访问权限
12         public void shout() {
13         super.shout();      //调用父类中的shout()方法
14                         System.out.println("汪汪汪……");
15         }
16     public void printName(){
17         System.out.println("名字:"+super.name);      //调用父类中的name属性
18     }
19 }
```

4.1.3 super关键字

```
20 // 定义测试类
21 public class Example05 {
22         public static void main(String[] args) {
23                 Dog dog = new Dog();  // 创建Dog类的实例对象
24                 dog.shout();          // 调用dog重写的shout()方法
25         dog.printName();       // 调用Dog类中的printName()方法
26         }
27 }
```

4.1.3 super关键字

程序运行结果如下图。



4.1.3 super关键字

上述代码中，第2~8行代码定义了一个Animal类，并在Animal类中定义了name属性和shout()方法。第10~19行代码定义了Dog类并继承了Animal类。在Dog类的shout()方法中使用“super.shout()”调用了父类被重写的shout()方法。在printName()方法中使用“super.name”访问父类的成员变量name。从运行结果中可以看出，子类通过super关键字可以成功地访问父类成员变量和成员方法。

4.1.3 super关键字

（2）使用super关键字访问父类中指定的构造方法，具体格式如下：

super(参数1,参数2...)

接下来就通过一个案例学习如何使用super关键字调用父类的构造方法。

```
1 // 定义Animal类
2 class Animal {
3    private String name;
4    private int age;
5    public Animal(String name, int age) {
6        this.name = name;
7        this.age = age;
8    }
```

4.1.3 super关键字

```
9 public String getName() {
10        return name;
11    }
12    public void setName(String name) {
13        this.name = name;
14    }
15    public int getAge() {
16        return age;
17    }
18    public void setAge(int age) {
19        this.age = age;
20    }
```

```
21 public String info() {
22        return "名称："+this.getName()+",年龄：  "+this.getAge();
23    }
24 }
25 // 定义Dog类继承动物类
26 class Dog extends Animal {
27    private String color;
28    public Dog(String name, int age, String color) {
29        super(name, age);
30        this.setColor(color);
31    }
32    public String getColor() {
33        return color;
34    }
```

4.1.3 super关键字

```
35 public void setColor(String color) {
36        this.color = color;
37    }
38    //重写父类的info()方法
39    public String info() {
40        return super.info()+",颜色："+this.getColor();    //扩充父类中的方法
41    }
42 }
43 // 定义测试类
44 public class Example06 {
45    public static void main(String[] args) {
46        Dog dog = new Dog("牧羊犬",3,"黑色"); // 创建Dog类的实例对象
47        System.out.println(dog.info());
48    }
49 }
```

4.1.3 super关键字

程序运行结果如下图。

4.1.3 super关键字

上述代码中，第29行代码使用super()调用了父类中有两个参数的构造方法；39~41行代码是在子类Dog中重写了父类Animal中的info()方法；第46~47行代码是实例化了一个Dog对象并调用了info()方法。由运行结果可知，程序输出的内容是在子类中定义的内容。这说明，如果在子类中重写了父类的info()方法，使用子类的实例化对象调用info()方法时，会优先调用子类中的info()方法。

4.1.3 super关键字

通过super()调用父类构造方法的代码必须位于子类构造方法的第一行，并且只能出现一次。

4.1.3 super关键字

super与this关键字的作用非常相似，都可以调用构造方法、普通方法和属性，但是两者之间还是有区别的，super与this的区别如下表。

区别点	this	super
属性访问	访问本类中的属性，如果本类中没有该属性，则从父类中查找。	访问父类中的属性
方法	访问本类中的方法，如果本类中没有该方法，则从父类中继续查找	直接访问父类中的方法
调用构造	调用本类构造，必须放在构造方法的首行	调用父类构造，必须放在子类构造方法的首行

4.1.3 super关键字

this和super两者不可以同时出现，因为this和super在调用构造方法时都要求必须放在构造方法的首行。

4.2 final关键字

final的英文意思是“最终”。在Java中，可以使用final关键字声明类、属性、方法，在声明时需要注意以下几点：

（1）使用final修饰的类不能有子类。

（2）使用final修饰的方法不能被子类重写。

（3）使用final修饰的变量（成员变量和局部变量）是常量，常量不可修改。

4.2.1 final关键字修饰类

Java中的类被final关键字修饰后，该类将不可以被继承，即不能够派生子类。

```
1 // 使用final关键字修饰Animal类
2 final class Animal {
3             // 方法体为空
4 }
5 // Dog类继承Animal类
6 class Dog extends Animal {
7             // 方法体为空
8 }
9 // 定义测试类
10 public class Example07 {
11            public static void main(String[] args) {
12                        Dog dog = new Dog();              // 创建Dog类的实例对象
13            }
14 }
```

4.2.1 final关键字修饰类

程序编译时报错，如下图。

4.2.2 final关键字修饰方法

当一个类的方法被final关键字修饰后，这个类的子类将不能重写该方法。接下来通过一个案例验证。

```
1 // 定义Animal类
2 class Animal {
3     // 使用final关键字修饰shout()方法
4         public final void shout() {
5         }
6 }
7 // 定义Dog类继承Animal类
8 class Dog extends Animal {
9     // 重写Animal类的shout()方法
10        public void shout() {
11        }
12 }
```

4.2.2 final关键字修饰方法

```
13 // 定义测试类
14 public class Example08 {
15         public static void main(String[] args) {
16                 Dog dog=new Dog(); // 创建Dog类的实例对象
17         }
18 }
```

4.2.2 final关键字修饰方法

程序运行结果如下图。

4.2.2 final关键字修饰方法

在上述代码中，第10行代码在Dog类中重写了父类Animal中的shout()方法，编译报错。这是因为Animal类的shout()方法被final修饰，而被final关键字修饰的方法为最终方法，子类不能对该方法进行重写。因此，当在父类中定义某个方法时，如果不希望被子类重写，就可以使用final关键字修饰该方法。

4.2.3 final关键字修饰变量

Java中被final修饰的变量是为常量，常量只能在声明时被赋值一次，在后面的程序中，其值不能被改变。如果再次对该常量赋值，则程序会在编译时报错。接下来通过一个案例进行验证。

```
1 public class Example09 {
2         public static void main(String[] args) {
3                 final int AGE = 18;    // 第一次可以赋值
4                 AGE = 20;              // 再次赋值会报错
5         }
6 }
```

4.2.3 final关键字修饰变量

编译上述程序，程序报错，如下图。

4.2.3 final关键字修饰变量

在使用final声明变量时，要求全部的字母大写。

如果一个程序中的变量使用public static final声明，则此变量将成为全局变量，如下面代码所示。

```
public static final String NAME = "哈士奇";
```

4.3.1 抽象类

当定义一个类时，常常需要定义一些成员方法描述类的行为特征，但有时这些方法的实现方式是无法确定的。例如，前面在定义Animal类时，shout()方法用于描述动物的叫声，但是针对不同的动物，叫声也是不同的，因此在shout()方法中无法准确描述动物的叫声。

4.3.1 抽象类

针对上面描述的情况，Java提供了抽象方法来满足这种需求。抽象方法是使用abstract关键字修饰的成员方法，抽象方法在定义时不需要实现方法体。抽象方法的定义格式如下：

abstract void 方法名称(参数);

当一个类包含了抽象方法，该类必须是抽象类。抽象类和抽象方法一样，必须使用abstract关键字进行修饰。

4.3.1 抽象类

抽象类的定义格式如下：

```
abstract class 抽象类名称{
    访问权限 返回值类型 方法名称(参数){
        return [返回值];
    }
    访问权限 abstract 返回值类型 抽象方法名称(参数);     //抽象方法，无方法体
}
```

4.3.1 抽象类

抽象类的定义规则如下：

（1）包含一个以上抽象方法的类必须是抽象类。

（2）抽象类和抽象方法都要使用abstract关键字声明。

（3）抽象方法只需声明而不需要实现。

（4）如果一个类继承了抽象类，那么该子类必须实现抽象类中的全部抽象方法。

4.3.1 抽象类

接下来通过一个案例学习抽象类的使用。

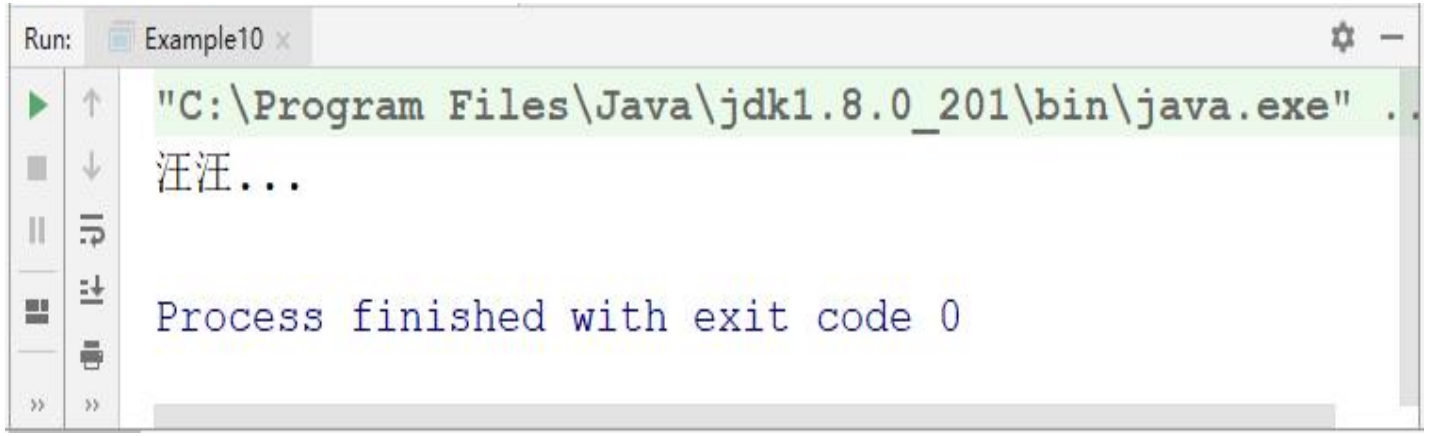
```
1 // 定义抽象类Animal
2 abstract class Animal {
3     // 定义抽象方法shout()
4             abstract void shout();
5 }
6 // 定义Dog类继承抽象类Animal
7 class Dog extends Animal {
8     // 实现抽象方法shout()
9             void shout() {
10                        System.out.println("汪汪...");
11            }
12 }
```

4.3.1 抽象类

```
13 // 定义测试类
14 public class Example10 {
15         public static void main(String[] args) {
16                 Dog dog = new Dog();    // 创建Dog类的实例对象
17                 dog.shout();                    // 调用dog对象的shout()方法
18         }
19 }
```

4.3.1 抽象类

程序运行结果如下图。



4.3.1 抽象类

上述代码中，第2~5行代码是声明了一个抽象类Animal，并在Animal类中声明了一个抽象方法shout()；第9~11行代码在子类Dog中实现父类Animal的抽象方法shout()；第17行代码通过子类的实例化对象调用shout()方法。

4.3.1 抽象类

使用abstract关键字修饰的抽象方法不能使用private修饰，因为抽象方法必须被子类实现，如果使用了private声明，则子类无法实现该方法。

4.3.2 接口

如果一个抽象类的所有方法都是抽象的，则可以将这个类定义接口。接口是Java中最重要的概念之一，接口是一种特殊的类，由全局常量和公共的抽象方法组成，不能包含普通方法。

在JDK8之前，接口是由全局常量和抽象方法组成的，且接口中的抽象方法不允许有方法体。JDK 8对接口进行了重新定义，接口中除了抽象方法外，还可以有默认方法和静态方法(也叫类方法)，默认方法使用default修饰，静态方法使用static修饰，且这两种方法都允许有方法体。

4.3.2 接口

接口使用interface关键字声明，语法格式如下：

```
public interface 接口名 extends 接口1,接口2... {
        public static final 数据类型 常量名 = 常量值;
        public abstract 返回值类型 抽象方法名(参数列表);
    public default 返回值类型 方法名(参数列表){
      //默认方法的方法体
    }
    public static 返回值类型方法名(参数列表){
      //类方法的方法体
    }
}
```

4.3.2 接口

在上述语法中，“extends 接口1,接口2...”表示一个接口可以有多个父接口，父接口之间使用逗号分隔。Java使用接口的目的是为了克服单继承的限制，因为一个类只能有一个父类，而一个接口可以同时继承多个父接口。接口中的变量默认使用“public static final”进行修饰，即全局常量。接口中定义的方法默认使用“public abstract”进行修饰，即抽象方法。如果接口声明为public，则接口中的变量和方法全部为public。

4.3.2 接口

在很多的Java程序中，经常看到编写接口中的方法时省略了public ，有很多读者认为它的访问权限是default，这实际上是错误的。不管写不写访问权限，接口中的方法访问权限永远是public。与此类似，在接口中定义常量时，可以省略前面的“public static final”，此时，接口会默认为常量添加“public static final”。

4.3.2 接口

与抽象类一样，接口的使用必须通过子类，子类通过implements关键字实现接口，并且子类必须实现接口中的所有抽象方法。需要注意的是，一个类可以同时实现多个接口，多个接口之间需要使用英文逗号(,)分隔。

定义接口的实现类，语法格式如下：

```
修饰符 class 类名 implements 接口1,接口2,...{
    ...
}
```

4.3.2 接口

下面通过一个案例学习接口的使用。

```
1 // 定义抽象类Animal
2 interface Animal {
3    int ID = 1;                          // 定义全局常量
4    String NAME = "牧羊犬";
5    void shout();                        // 定义抽象方法shout()
6    static int getID(){
7        return Animal.ID;
8    }
9    public void info();                  // 定义抽象方法info()
10 }
```

4.3.2 接口

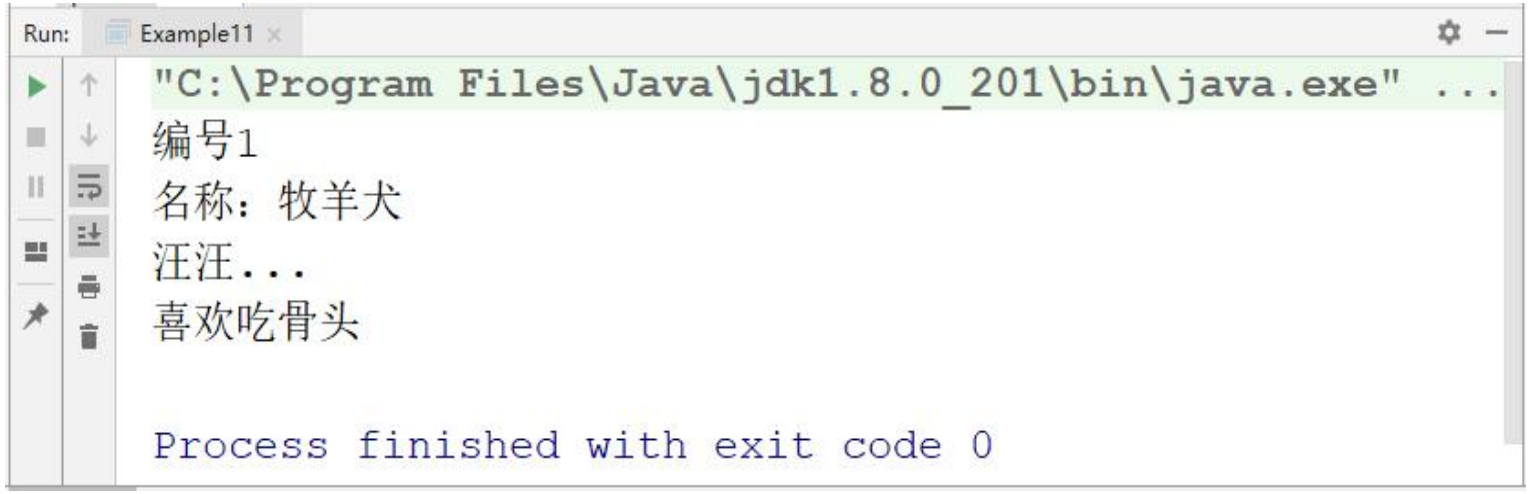
```
11 interface Action {
12    public void eat();                          // 定义抽象方法eat()
13 }
14 // 定义Dog类实现Animal接口和Action接口
15 class Dog implements Animal,Action{
16   // 重写Action接口中的抽象方法eat()
17    public void eat() {
18        System.out.println("喜欢吃骨头");
19    }
20    // 重写Animal接口中的抽象方法shout()
21    public void shout() {
22        System.out.println("汪汪...");
23    }
```

4.3.2 接口

```
24 // 重写Animal接口中的抽象方法info()
25    public void info() {
26        System.out.println("名称："+NAME);
27    }
28 }
29 // 定义测试类
30 class Example11 {
31  public static void main(String[] args) {
32        System.out.println("编号"+Animal.getID());
33        Dog dog = new Dog();        // 创建Dog类的实例对象
34        dog.info();
35        dog.shout();                // 调用Dog类中重写的shout()方法
36        dog.eat();                  // 调用Dog类中重写的eat()方法
37    }}
```

4.3.2 接口

程序运行结果如下图。



4.3.2 接口

上述代码中，第2~10行代码定义了一个Animal接口，在Animal接口中定义了全局常量id和NAME、抽象方法shout()、info()和静态方法getID()。第11~13行代码定义了一个Action接口，在Action接口中定义了一个抽象方法eat()。第15~28行代码定义了一个Dog类，Dog类通过implements实现了Animal接口和Action接口，并实现了这两个接口中的方法。第32行代码使用Animal接口名直接访问了Animal接口中的静态方法getID()。第33~36行代码创建并实例化了Dog类对象dog，通过dog对象访问了Animal接口和Action接口中的常量、抽象方法 。

4.3.2 接口

从运行结果可以看出，Dog类的实例化对象可以访问接口中的常量、实现的接口方法以及本类内部的方法，而接口中的静态方法则可以直接使用接口名调用。需要注意的是，接口的实现类，必须实现接口中的所有方法，否则程序编译报错。

4.3.2 接口

如果在开发中一个子类既要实现接口又要继承抽象类，则可以按照以下格式定义子类。

修饰符class 类名 extends 父类名implements 接口1,接口2,... {

    ...

}

4.3.2 接口

下面通过一个案例演示子类既实现接口又继承抽象类的情况。

```
1 // 定义抽象类Animal
2 interface Animal {
3    public String NAME = "牧羊犬";
4    public void shout();              // 定义抽象方法shout()
5    public void info();               // 定义抽象方法info()
6 }
7 abstract class Action {
8    public abstract void eat();       // 定义抽象方法eat()
9 }
```

4.3.2 接口

```
10 // 定义Dog类实现Animal接口和Action接口
11 class Dog extends Animal implements Action{
12    // 重写Action接口中的抽象方法eat()
13    public void eat() {
14        System.out.println("喜欢吃骨头");
15    }
16    // 重写Animal接口中的抽象方法shout()
17    public void shout() {
18        System.out.println("汪汪...");
19    }
20    // 重写Animal接口中的抽象方法info()
21    public void info() {
22      System.out.println("名称："+NAME);
23    }
24 }
```

4.3.2 接口

```
25 // 定义测试类
26 class Example12 {
27    public static void main(String[] args) {
28        Dog dog = new Dog();    // 创建Dog类的实例对象
29        dog.info();             // 调用Dog类中重写的info()方法
30        dog.shout();            // 调用Dog类中重写的shout()方法
31        dog.eat();                          // 调用Dog类中重写的eat()方法
32    }
33 }
```

4.3.2 接口

上述代码中，Dog类通过extends实现了Animal接口，同时通过implements实现了抽象类Action。因为Animal接口和抽象类Action本身都有抽象方法，所以子类中必须实现。

4.3.2 接口

在Java中，接口是不允许继承抽象类的，但是允许一个接口继承多个接口。接下来通过一个案例讲解接口的继承。

```
1 // 定义抽象类Animal
2 interface Animal {
3    public String NAME = "牧羊犬";
4    public void info();          // 定义抽象方法info()
5 }
6 interface Color {
7    public void black();         // 定义抽象方法shout()
8 }
9 interface Action extends Animal,Color{
10    public void shout();                    // 定义抽象方法
black()
11 }
```

4.3.2 接口

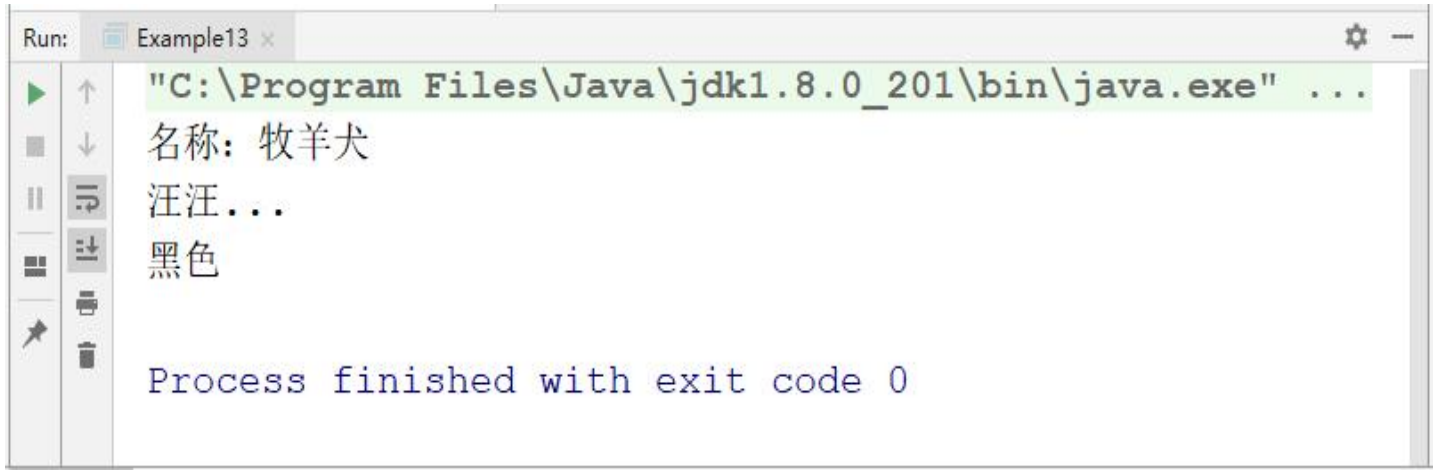
```
12 // 定义Dog类实现Action接口
13 class Dog implements Action{
14    // 重写Animal接口中的抽象方法info()
15    public void info() {
16        System.out.println("名称："+NAME);
17    }
18    // 重写Color接口中的抽象方法black()
19    public void black() {
20        System.out.println("黑色");
21    }
22    // 重写Action接口中的抽象方法shout()
23    public void shout() {
24        System.out.println("汪汪...");
25    }
26 }
```

4.3.2 接口

```
27 // 定义测试类
28 class Example13 {
29     public static void main(String[] args) {
30         Dog dog = new Dog(); // 创建Dog类的实例对象
31         dog.info();     // 调用Dog类中重写的info()方法
32         dog.shout();    // 调用Dog类中重写的shout()方法
33         dog.black();    // 调用Dog类中重写的eat()方法
34     }
35 }
```

4.3.2 接口

程序运行结果如下图。



4.3.2 接口

在上述代码中，第9~11行代码定义了接口Action并继承接口Animal和Color，这样接口Action中就同时拥有Animal和Color接口中info()和black()方法以及本类中shout()方法。在第13~26行代码定义了一个Dog类并实现了Action接口，这样Dog类就必须同时实现这3个抽象方法。

【案例4-1】打印不同的图形

本案例要求编写一个程序，可以根据用户要求在控制台打印出不同的图形。例如，用户自定义半径的圆形和用户自定义边长的正方形。

【案例4-2】饲养员喂养动物

饲养员在给动物喂食时，给不同的动物喂不同的食物，而且在每次喂食时，动物都会发出欢快的叫声。例如，给小狗喂骨头，小狗会汪汪叫；给小猫喂食，小猫会喵喵叫。

本案例要求编写一个程序模拟饲养员喂食动物的过程，案例要求如下：

（1）饲养员给小狗喂骨头，小狗汪汪叫。

（2）饲养员给小猫喂小鱼，小猫喵喵叫。

【案例4-3】多彩的声音

狗是人类最忠诚的伙伴，当狗遇到不同的人会有不同的反应，下面要求编写一个案例模拟狗遇到不同的人时的反应。首先需要有一只狗，不同的人去喂狗，狗会有不同的反应，具体如下：

（1）主人喂狗时，狗会很开心的欢跳.

（2）主人的家人喂狗时，狗会很平常的吃东西。

（3）陌生人喂狗时，狗会很有警惕，并发出叫声。

本案例要求使用抽象类实现。

【案例4-4】学生和老师

在班级中上课时，老师在讲台上讲课，偶有提问，会点名学生回答问题。虽然老师和学生都在讲话，但讲话的具体内容却不相同。本案例要求使用抽象类的知识编写一个程序实现老师上课的情景。

【案例4-5】图形的面积与周长计算程序

长方形和圆形都属于几何图形，都有周长和面积，并且它们都有自己的周长和面积计算公式。使用抽象类的知识设计一个程序，可以计算不同图形的面积和周长。

【案例4-6】研究生薪资管理

在学校中，学生每个月需要交相应的生活费，老师每个月有相应的工资，而在职研究生既是老师又是学生，所以在职研究生既需要交学费又会有工资。下面要求编写一个程序来统计在职研究生的收入与学费，如果收入减去学费不足2000 元，则输出“provide a loan”(需要贷款)信息。本案例要求使用接口实现该程序。

4.4.1 多态概述

多态性是面向对象思想中的一个非常重要的概念，在Java中，多态是指不同对象在调用同一个方法时表现出的多种不同行为。例如，要实现一个动物叫的方法，由于每种动物的叫声是不同的，因此可以在方法中接收一个动物类型的参数，当传入猫类对象时就发出猫类的叫声，传入犬类对象时就发出犬类的叫声。在同一个方法中，这种由于参数类型不同而导致执行效果不同的现象就是多态。

4.4.1 多态概述

Java中多态主要有以下两种形式：

（1）方法的重载。

（2）对象的多态性（方法重写）。

4.4.1 多态概述

接下来通过一个案例演示Java程序中的多态。

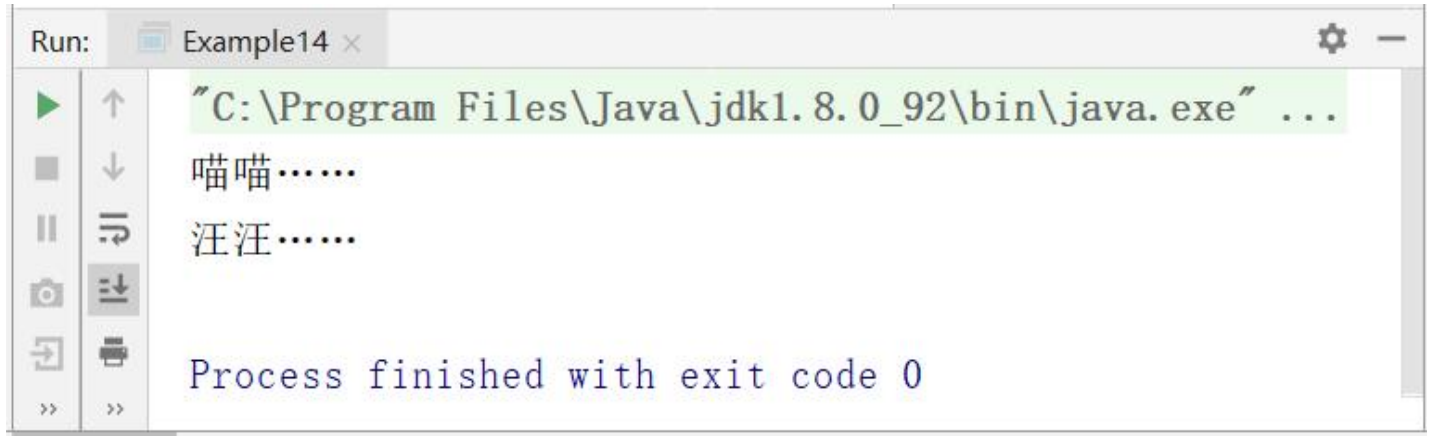
```
1 // 定义抽象类Anmal
2 abstract class Animal {
3  abstract  void shout();       // 定义抽象shout()方法
4 }
5 // 定义Cat类继承Animal抽象类
6 class Cat extends Animal {
7 // 实现shout()方法
8            public void shout() {
9                        System.out.println("喵喵……");
10           }
11 }
```

4.4.1 多态概述

```
1 // 定义Dog类继承Animal抽象类
2 class Dog extends Animal {
3     // 实现shout()方法
4             public void shout() {
5                     System.out.println("汪汪……");
6             }
7 }
8 // 定义测试类
9 public class Example14 {
10            public static void main(String[] args) {
11                    Cat an1 = new Cat(); // 创建Cat对象,使用Animal类型的变量an1引用
12                    Dog an2 = new Dog(); // 创建Dog对象,使用Animal类型的变量an2引用
13                    an1.shout();
14                    an2.shout();
15            }
16 }
```

4.4.1 多态概述

程序运行结果如下图。



4.4.1 多态概述

上述代码中，第2~4行代码定义了一个抽象类Animal，在抽象类Animal中定义了抽象方法shout()。第6~18行代码定义了两个继承Animal的类Cat和Dog，并在Cat类和Dog类中重写了Animal类中的shout()方法。第22~25行代码是创建了Cat类对象和Dog类对象，并将Cat类对象和Dog类对象向上转型成了Animal类型的对象，然后通过Animal类型的对象an1和an2调用shout()方法。从运行结果可以看出，对象an1和an2调用的分别是Cat类和Dog类中的shout()方法。

4.4.2 对象类型的转换

对象类型转换主要分为以下两种情况：

（1）向上转型：子类对象→父类对象。

（2）向下转型：父类对象→子类对象。

对于向上转型，程序会自动完成，而向下转型时，必须指明要转型的子类类型。

4.4.2 对象类型的转换

对象类型的转换格式如下所示。

对象向上转型：父类类型 父类对象 = 子类实例;

对象向下转型：

父类类型 父类对象 = 子类实例;

子类类型 子类对象 = （子类）父类对象;

4.4.2 对象类型的转换

下面通过一个案例介绍如何进行对象的向上转型操作。

```
1 // 定义类Anmal
2 class Animal {
3   public void shout(){
4    System.out.println("喵喵……");
5  }
6 }
```

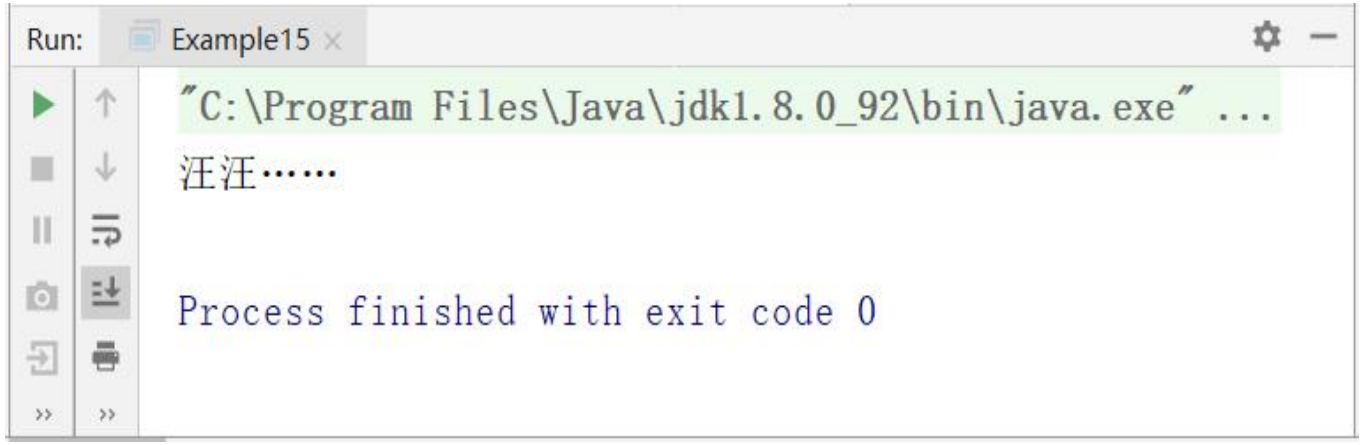
```
7 // Dog类
8 class Dog extends Animal {
9   // 重写shout()方法
10   public void shout() {
11       System.out.println("汪汪……");
12  }
13   public void eat() {
14       System.out.println("吃骨头……");
15  }
16 }
```

4.4.2 对象类型的转换

```
17 // 定义测试类
18 public class Example15 {
19   public static void main(String[] args) {
20       Dog dog = new Dog(); // 创建Dog对象
21       Animal an = dog;
22       an.shout();
23   }
24 }
```

4.4.2 对象类型的转换

程序运行结果如下图。



4.4.2 对象类型的转换

上述代码中，第20~22行代码是创建了一个dog对象，并将dog对象向上转型成Animal类型的对象an，然后使用对象an调用shout()方法。从程序的运行结果中可以发现，虽然是使用父类对象an调用了shout()方法，但实际上调用的是被子类重写过的shout()方法。也就是说，如果对象发生了向上转型关系后，所调用的方法一定是被子类重写过的方法。

4.4.2 对象类型的转换

父类Animal的对象an是无法调用Dog类中的eat()方法的，因为eat()方法只在子类中定义，而没有在父类中定义。

4.4.2 对象类型的转换

在进行对象的向下转型前，必须发生对象向上转型，否则将出现对象转换异常。接下来通过一个案例演示对象进行向下转型。

```
1 // 定义类Anmal
2 class Animal {
3   public void shout(){
4    System.out.println("喵喵……");
5  }
6 }
```

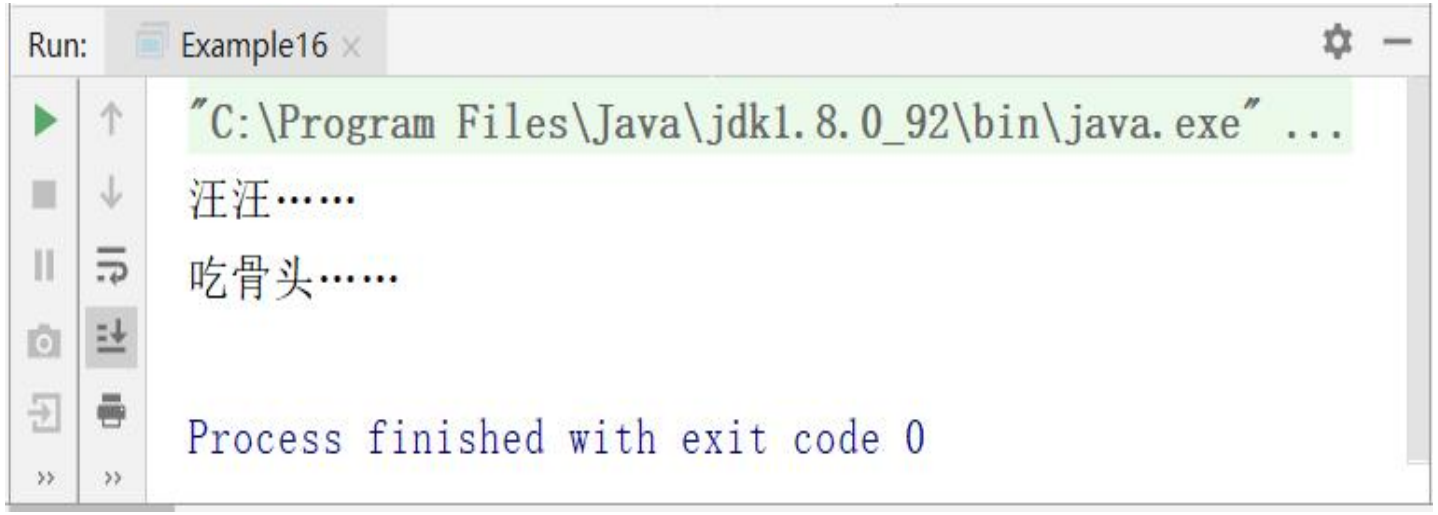
```
7 // Dog类
8 class Dog extends Animal {
9   // 重写shout()方法
10   public void shout() {
11       System.out.println("汪汪……");
12  }
13   public void eat() {
14       System.out.println("吃骨头……");
15  }
16 }
```

4.4.2 对象类型的转换

```
17 // 定义测试类
18 public class Example16 {
19   public static void main(String[] args) {
20       Animal an = new Dog();  // 此时发生了向上转型，子类→父类
21       Dog dog = (Dog)an;      // 此时发生了向下转型
22       dog.shout();
23       dog.eat();
24   }
25 }
```

4.4.2 对象类型的转换

程序运行结果如下图。



4.4.2 对象类型的转换

在上述程序中，第20行代码发生了向上转型，将Dog类的实例转换成了Animal类的实例an，第21行代码是将Animal类的实例转换为Dog类的实例。第22行代码使用dog对象调用shout()方法，由于Animal类的shout()方法已被子类Dog类重写，因此dog对象调用的方法是被子类重写过的方法。

4.4.2 对象类型的转换

在向下转型时，不能直接将父类实例强制转换为子类实例，否则程序会报错。例如，将文件4-16中的第20~21行代码修改为下面一行代码，则程序报错。

```
Dog dog = (Dog)new Animal(); //编译错误
```

4.4.3 instanceof关键字

Java中可以使用instanceof关键字判断一个对象是否是某个类（或接口）的实例，语法格式如下所示。

对象 instanceof类（或接口）

在上述格式中，如果对象是指定的类的实例对象，则返回true，否则返回false。

4.4.3 instanceof关键字

接下来通过一个案例演示instanceof关键字的用法。

```
1 // 定义类Anmal
2 class Animal {
3    public void shout(){
4     System.out.println("动物叫……");
5  }
6 }
```

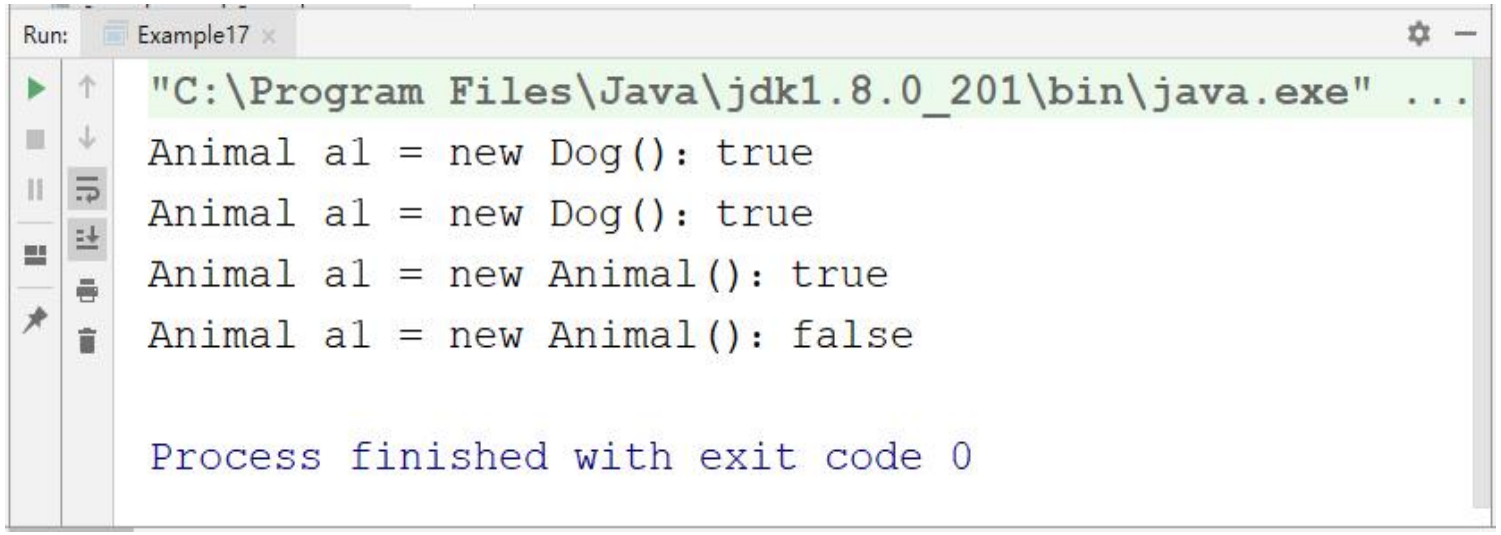
```
7 // Dog类
8 class Dog extends Animal {
9    // 重写shout()方法
10    public void shout() {
11          System.out.println("汪汪……");
12   }
13    public void eat() {
14          System.out.println("吃骨头……");
15   }
16}
```

4.4.3 instanceof关键字

```
17 // 定义测试类
18 public class Example17 {
19  public static void main(String[] args) {
20          Animal a1 = new Dog();        // 通过向上转型实例化Animal对象
21          System.out.println("Animal a1 = new Dog() : "+(a1 instanceof Animal));
22          System.out.println("Animal a1 = new Dog() : "+(a1 instanceof Dog));
23          Animal a2 = new Animal();        // 实例化Animal对象
24          System.out.println("Animal a1 = new Animal() : "+(a2 instanceof Animal));
25          System.out.println("Animal a1 = new Animal() : "+(a2 instanceof Dog));
26   }
27 }
```

4.4.3 instanceof关键字

程序运行结果如下图。



4.4.3 instanceof关键字

上述代码中，第2~6行代码定义了Animal类；第8~16行代码定义了Dog类继承Animal类；第20行代码实例化Dog类对象，并将Dog类实例向上转型为Animal类对象a1。第21行代码是通过instanceof关键字判断对象a1是否是Animal类的实例，第22行代码是通过instanceof关键字判断对象a1是否是Dog类的实例；第23行代码是实例化了一个Animal类对象a2，第24行代码是通过instanceof关键字判断对象a2是否是Animal类的实例，第25行代码通过instanceof关键字判断对象a2是否是Dog类的实例。

【案例4-7】经理与员工工资案例

某公司的人员分为员工和经理两种，但经理也属于员工中的一种，公司的人员都有自己的姓名和地址，员工和经理都有自己的工号、工资、工龄等属性，但经理不同员工的是，经理有自己在公司对应的级别。假设每次给员工涨工资一次能涨10，经理能涨20%。本案例要求利用多态实现给员工和经理涨工资。

【案例4-8】模拟物流快递系统

网购已成为人们生活的重要组成部分，当人们在购物网站中下订单后，订单中的货物就会在经过一系列的流程后，送到客户的手中。而在送货期间，物流管理人员可以在系统中查看所有物品的物流信息。编写一个模拟物流快递系统的程序，模拟后台系统处理货物的过程。

4.5 Object类

Java提供了一个Object类，它是所有类的父类，每个类都直接或间接继承Object类，因此Object类通常被称之为超类。

当定义一个类时，如果没有使用extends关键字为这个类显式地指定父类，那么该类会默认继承Object类。

4.5 Object类

Object类常用方法如下表。

方法名称	方法说明
boolean equals()	判断两个对象是否“相等”
int hashCode()	返回对象的哈希码值
String toString()	返回对象的字符串表示形式

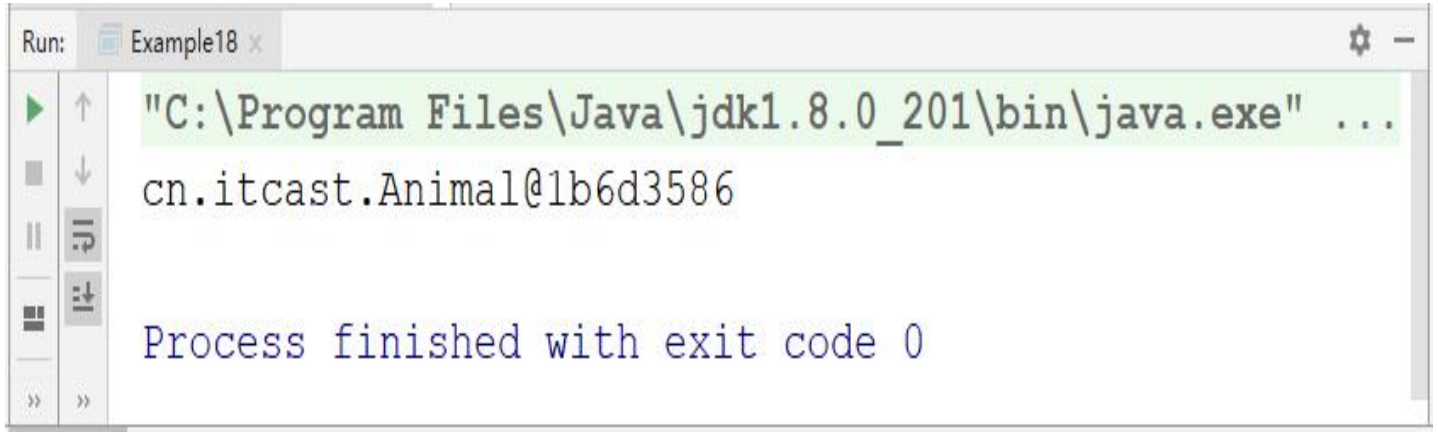
4.5 Object类

下面通过一个例子演示Object类中toString()方法的使用。

```
1 // 定义Animal类
2 class Animal {
3     // 定义动物叫的方法
4             void shout() {
5                     System.out.println("动物叫！");
6             }
7 }
8 // 定义测试类
9 public class Example18 {
10            public static void main(String[] args)  {
11                    Animal animal = new Animal();           // 创建Animal类对象
12                    System.out.println(animal.toString());          // 调用toString()方法并打印
13            }
14}
```

4.5 Object类

程序运行结果如下图。



4.5 Object类

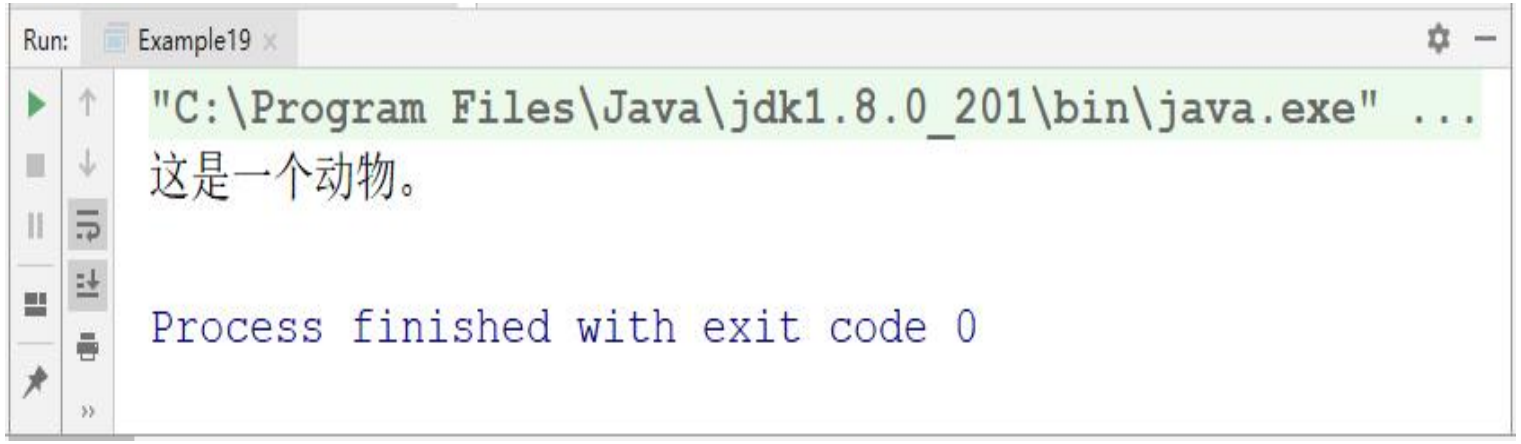
在实际开发中，通常希望对象的toString()方法返回的不仅仅是基本信息，而是对象特有的信息，这时可以重写Object类的toString()方法。

4.5 Object类

```
1 // 定义Animal类
2 class Animal {
3   //重写Object类的toString()方法
4   public String toString(){
5             return "这是一个动物。";
6   }
7 }
8 // 定义测试类
9 public class Example19 {
10            public static void main(String[] args) {
11                      Animal animal = new Animal();                    // 创建Animal类对象
12                      System.out.println(animal.toString()); // 调用toString()方法并打印
13            }
14 }
```

4.5 Object类

程序运行结果如下图。



4.6 内部类

在Java中，允许在一个类的内部定义类，这样的类称作内部类，内部类所在的类称作外部类。在实际开发中，根据内部类的位置、修饰符和定义方式的不同，内部类可分为4种，分别是成员内部类、局部内部类、静态内部类、匿名内部类。

4.6.1 成员内部类

在一个类中除了可以定义成员变量、成员方法，还可以定义类，这样的类被称作成员内部类。成员内部类可以访问外部类的所有成员。

4.6.1 成员内部类

接下来通过一个案例学习如何定义成员内部类。

```
1 class Outer {
2    int m = 0;                              // 定义类的成员变量
3   // 下面的代码定义了一个成员方法，方法中访问内部类
4    void test1() {
5       System.out.println("外部类成员方法");
6    }
7    // 下面的代码定义了一个成员内部类
8    class Inner {
9        int n = 1;
10        void show1() {
11           // 在成员内部类的方法中访问外部类的成员变量
12           System.out.println("外部成员变量m = " + m);
13       }
```

4.6.1 成员内部类

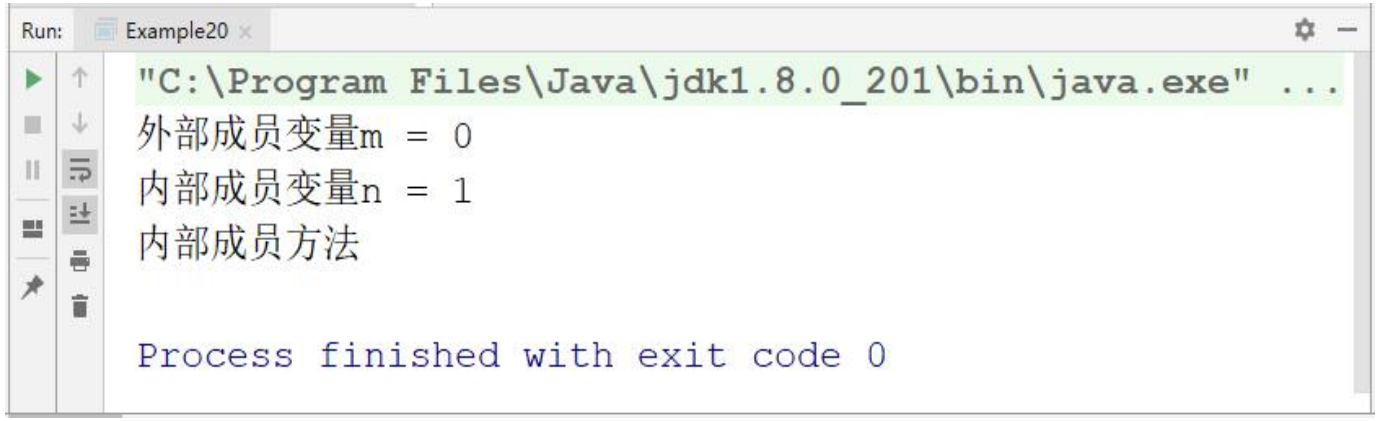
```
14 void show2() {
15          // 在成员内部类的方法中访问外部类的成员变量
16          System.out.println("内部成员方法");
17      }
18  }
19  void test2() {
20      Inner inner = new Inner();
21      System.out.println("内部成员变量n = " + inner.n);
22      inner.show2();
23  }
24 }
```

4.6.1 成员内部类

```
25 public class Example20 {
26    public static void main(String[] args) {
27        Outer outer = new Outer();
28        Outer.Inner inner = outer.new Inner();
29        inner.show1();
30       outer.test2();
31    }
32 }
```

4.6.1 成员内部类

程序运行结果如下图。



4.6.1 成员内部类

上述代码中，第1~24行代码定义了一个Outer类，Outer类就是一个外部类。第8~23行代码是在Outer类内部定义了Inner类，Inner类就是Outer类的成员内部类。Outer类中定义了test1()和test2()两个方法。Inner类定义了一个show()方法，在show()方法中访问外部类的成员变量num；test2()方法中创建了内部类Inner的实例对象，并通过该对象调用show()方法，将num值输出。从运行结果可以看出，内部类可以在外部类中使用，并能访问外部类的成员。

4.6.1 成员内部类

如果想通过外部类访问内部类，则需要通过外部类创建内部类对象，创建内部类对象的具体语法格式如下：

外部类名.内部类名 变量名 = new 外部类名().new 内部类名();

4.6.2 局部内部类

局部内部类，也叫作方法内部类，是指定义在某个局部范围中的类，它和局部变量一样，都是在方法中定义的，有效范围只限于方法内部。

在局部内部类中，局部内部类可以访问外部类的所有成员变量和方法，而局部内部类中变量和方法却只能在所属方法中访问。

4.6.2 局部内部类

接下来通过一个案例学习局部内部类的定义和使用。

```
1 class Outer {
2     int m = 0;                    // 定义类的成员变量
3   // 下面的代码定义了一个成员方法，方法中访问内部类
4     void test1() {
5         System.out.println("外部类成员方法");
6   }
```

4.6.2 局部内部类

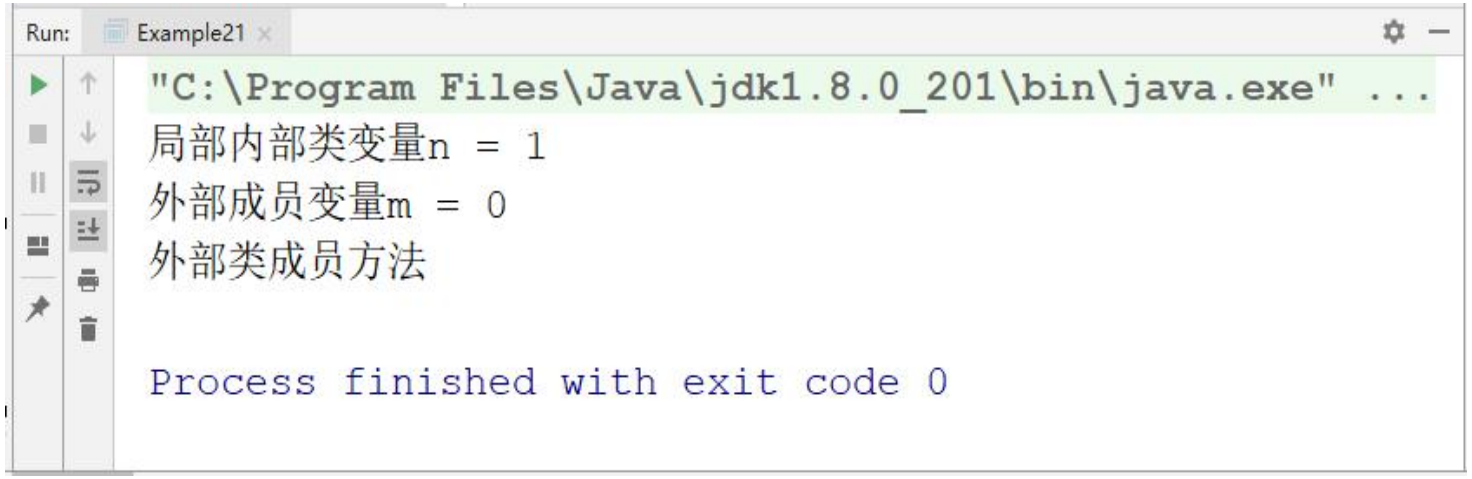
```
7 void test2() {
8       // 下面的代码定义了一个成员内部类
9       class Inner {
10          int n = 1;
11          void show() {
12              // 在成员内部类的方法中访问外部类的成员变量
13              System.out.println("外部成员变量m = " + m);
14              test1();
15          }
16      }
17      Inner inner = new Inner();
18      System.out.println("局部内部类变量n = " + inner.n);
19      inner.show();
20  }
21}
```

4.6.2 局部内部类

```
22 public class Example21 {
23     public static void main(String[] args) {
24         Outer outer = new Outer();
25         outer.test2();
26     }
27 }
```

4.6.2 局部内部类

程序运行结果如下图。



4.6.2 局部内部类

在上述代码中，第1~21行代码定义了一个外部类Outer，并在该类中定义了成员变量m、成员方法test1()和test2()。第9~15行代码是在外部类的成员方法test2()中定义了一个局部内部类Inner；然后在局部内部类Inner中，编写了show()方法。第13~14行代码是对外部类变量和方法的调用；第17~19行代码是在test2()方法中创建了局部内部类Inner对象，并调用局部内部类的方法和变量。

4.6.3 静态内部类

所谓静态内部类，就是使用static关键字修饰的成员内部类。与成员内部类相比，在形式上，静态内部类只是在内部类前增加了static关键字，但在功能上，静态内部类只能访问外部类的静态成员，通过外部类访问静态内部类成员时，可以跳过外部类直接访问静态内部类。

4.6.3 静态内部类

创建静态内部类对象的基本语法格式如下：

外部类名.静态内部类名 变量名 = new 外部类名().静态内部类名();

4.6.3 静态内部类

接下来通过一个案例学习静态内部类的定义和使用。

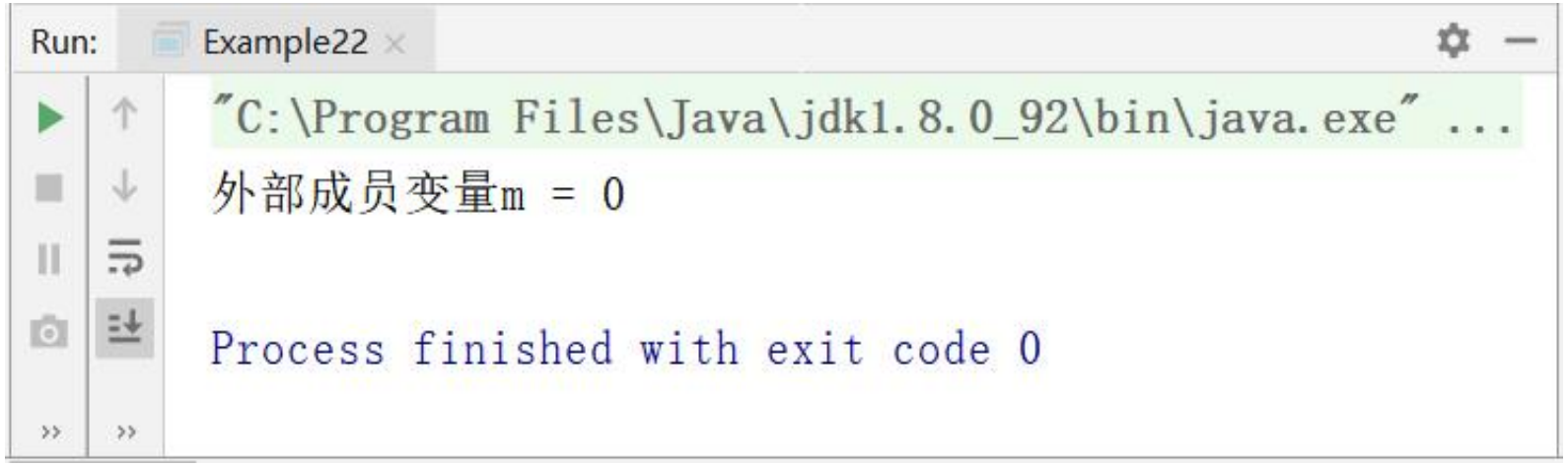
```
1 class Outer {
2    static int m = 0; // 定义类的成员变量
3    // 下面的代码定义了一个静态内部类
4    static class Inner {
5        int n = 1;
6        void show() {
7            // 在静态内部类的方法中访问外部类的成员变量
8            System.out.println("外部静态变量m = " + m);
9         }
10    }
11 }
```

4.6.3 静态内部类

```
12 public class Example22 {
13    public static void main(String[] args) {
14        Outer.Inner inner = new Outer.Inner();
15        inner.show();
16    }
17 }
```

4.6.3 静态内部类

程序运行结果如下图。



4.6.3 静态内部类

上述代码中，第1~11行代码定义了一个外部类Outer，其中第2~10行代码是在Outer类中定义了静态成员变量和静态内部类Inner。然后在静态内部类Inner中，编写了一个show()方法，在show()方法中打印了外部静态变量m，第14~15行代码是声明了一个内部类对象inner，并使用inner对象调用show()方法测试对外部类静态变量m的调用。

4.6.4 匿名内部类

匿名内部类是没有名称的内部类。在Java中调用某个方法时，如果该方法的参数是接口类型，除了可以传入一个接口实现类，还可以使用实现接口的匿名内部类作为参数，在匿名内部类中直接完成方法的实现。

创建匿名内部类的基本语法格式如下：

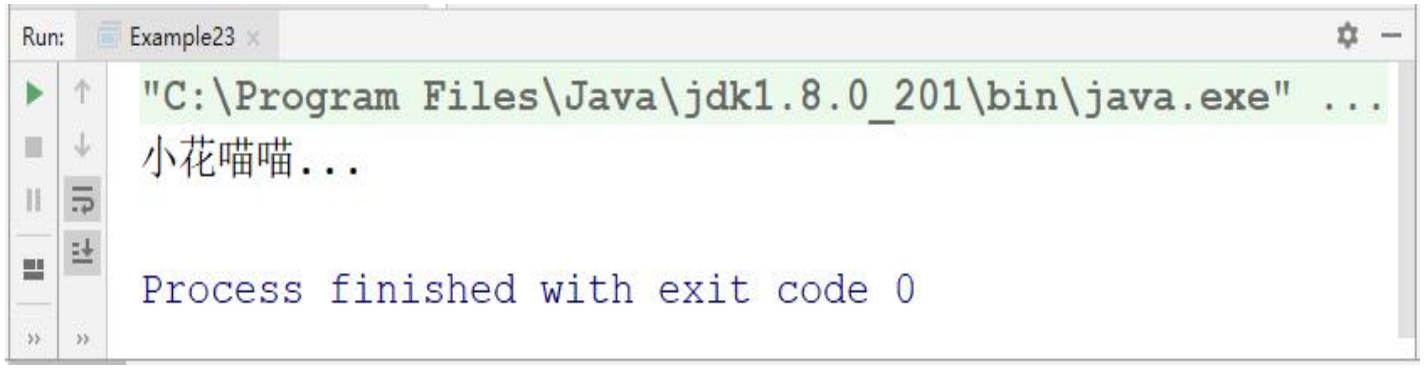
```
new 父接口(){
   //匿名内部类实现部分
}
```

4.6.4 匿名内部类

```
1 interface Animal{
2   void shout();
3 }
4 public class Example23{
5    public static void main(String[] args){
6       String name = "小花";
7       animalShout(new Animal(){
8          @Override
9           public void shout() {
10               System.out.println(name+"喵喵...");
11           }
12      });
13   }
14   public static void animalShout(Animal an){
15       an.shout();
16    }
17 }
```

4.6.4 匿名内部类

程序运行结果如下图。



4.6.4 匿名内部类

第1~3行代码创建了Animal接口；第7~12行代码是调用animalShout()方法，将实现Animal接口的匿名内部类作为animalShout()方法的参数，并在匿名内部类中重写了Animal接口的show()方法。

在上述代码中的匿名内部类中访问了局部变量name，而局部变量name并没有使用final修饰符修饰，程序也没有报错。这是JDK 8的新增特性，允许在局部内部类、匿名内部类中访问非final修饰的局部变量，而在JDK 8之前，局部变量前必须加final修饰符，否则程序编译时报错。

4.6.4 匿名内部类

匿名类的编写步骤：

（1）在调用animalShout()方法时，在方法的参数位置写上new Animal(){}，这相当于创建了一个实例对象，并将对象作为参数传给animalShout()方法。在new Animal()后面有一对大括号，表示创建的对象为Animal的子类实例，该子类是匿名的。具体代码如下所示：

```
animalShout(new Animal(){});
```

4.6.4 匿名内部类

（2）在大括号中编写匿名子类的实现代码，具体如下所示：

```
animalShout(new Animal() {
        public void shout() {
                System.out.println("喵喵……");
        }
});
```

4.7.1 什么是异常

尽管人人希望自己身体健康，处理的事情都能顺利进行，但在实际生活中总会遇到各种状况，比如感冒发烧，工作时电脑蓝屏、死机等。同样，在程序运行的过程中，也会发生各种非正常状况，例如，程序运行时磁盘空间不足、网络连接中断、被装载的类不存在等。针对这种情况，Java语言引入了异常，以异常类的形式对这些非正常情况进行封装，通过异常处理机制对程序运行时发生的各种问题进行处理。

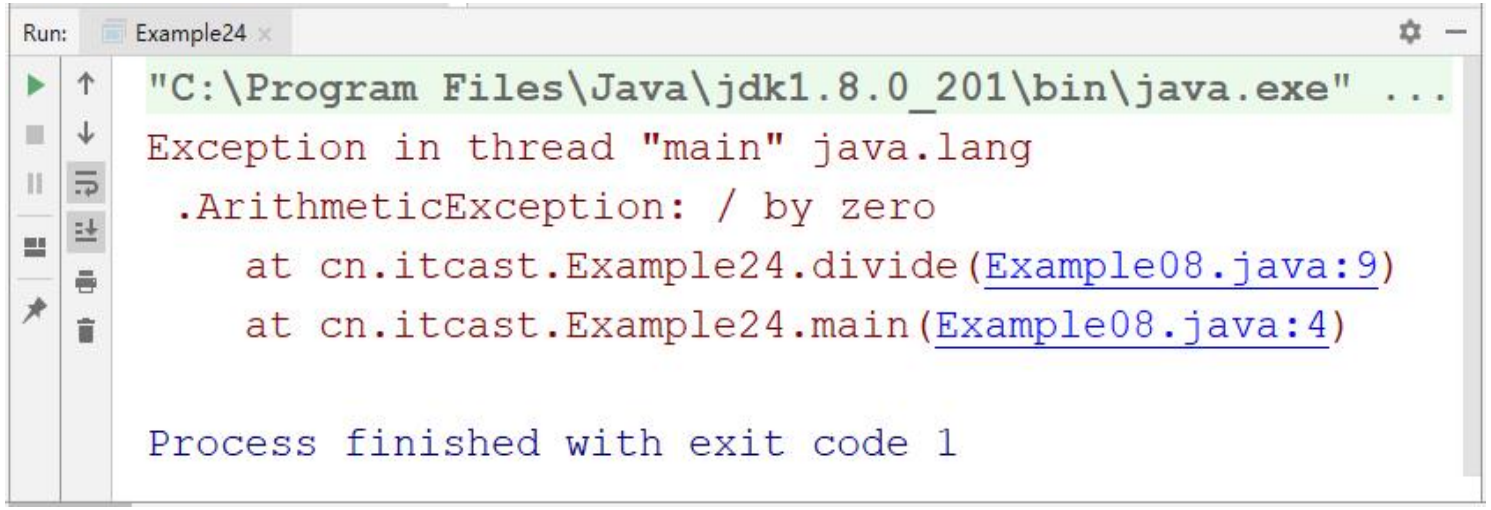
4.7.1 什么是异常

接下来通过一个案例认识一下什么是异常。

```
1 public class Example24 {
2         public static void main(String[] args) {
3                 int result = divide(4, 0);    // 调用divide()方法
4                 System.out.println(result);
5         }
6   //下面的方法实现了两个整数相除
7         public static int divide(int x, int y) {
8                 int result = x / y;             // 定义一个变量result记录两个数相除的结果
9                 return result;                  // 将结果返回
10        }
11 }
```

4.7.1 什么是异常

程序运行结果如下图。

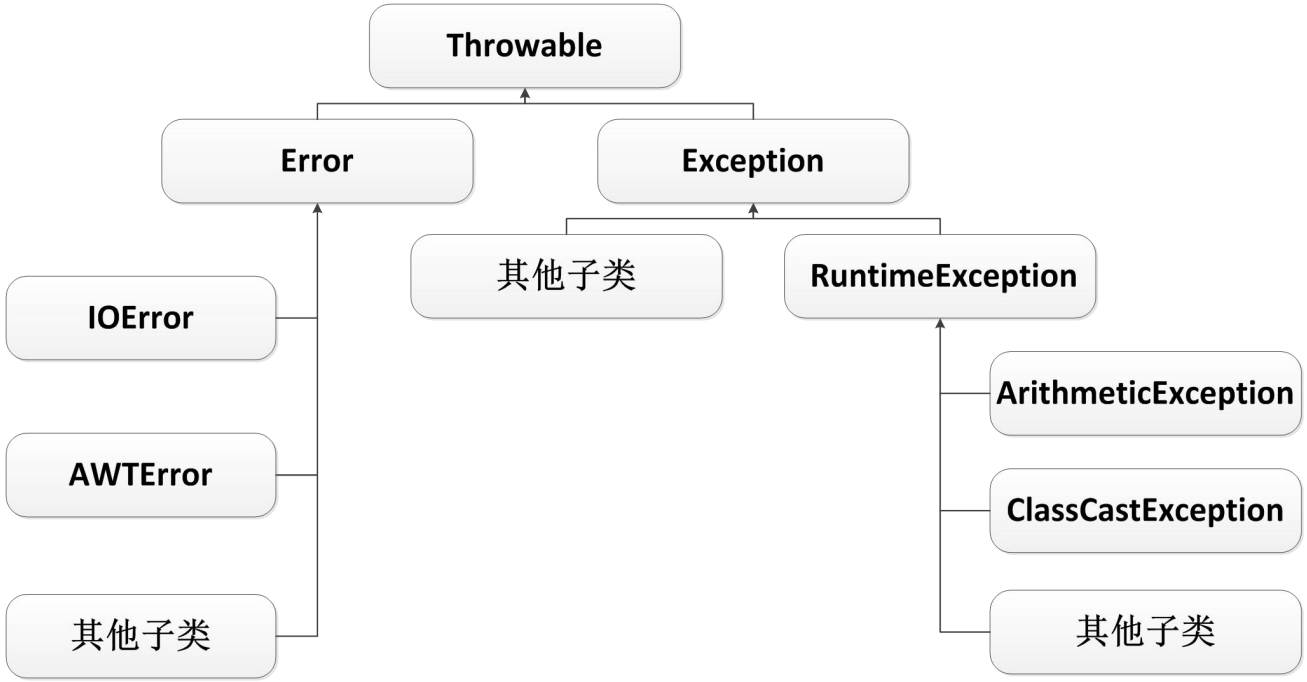


4.7.1 什么是异常

从运行结果可以看出，程序发生了算术异常（ArithmeticException），该异常是由于文件4-24中的第3行代码调用divide()方法时传入了参数0，运算时出现了被0除的情况。异常发生后，程序会立即结束，无法继续向下执行。

4.7.1 什么是异常

上述程序产生的ArithmeticException异常只是Java异常类中的一种，Java提供了大量的异常类，这些类都继承自java.lang.Throwable类。接下来通过一张图展示Throwable类的继承体系。



4.7.1 什么是异常

Throwable有两个直接子类Error和Exception，其中，Error代表程序中产生的错误，Exception代表程序中产生的异常。

- Error类称为错误类，它表示Java程序运行时产生的系统内部错误或资源耗尽的错误，这类错误比较严重，仅靠修改程序本身是不能恢复执行的。举一个生活中的例子，在盖楼的过程中因偷工减料，导致大楼坍塌，这就相当于一个Error。例如，使用java命令去运行一个不存在的类就会出现Error错误。

4.7.1 什么是异常

- Exception类称为异常类，它表示程序本身可以处理的错误，在Java程序中进行的异常处理，都是针对Exception类及其子类的。在Exception类的众多子类中有一个特殊的子类—RuntimeException类，RuntimeException类及其子类用于表示运行时异常。Exception类的其他子类都用于表示编译时异常。

4.7.1 什么是异常

Throwable类中的常用方法如下表。

方法声明	功能描述
String getMessage()	返回异常的消息字符串
String toString()	返回异常的简单信息描述
void printStackTrace()	获取异常类名和异常信息，以及异常出现在程序中的位置，把信息输出在控制台。

4.7.2 try…catch和finally

出现异常后，程序会立即终止。为了解决异常，Java提供了对异常进行处理的方式——异常捕获。异常捕获使用try…catch语句实现，try…catch具体语法格式如下：

```
try{
    //程序代码块
}catch(ExceptionType(Exception类及其子类) e){
    //对ExceptionType的处理
}
```

4.7.2 try...catch和finally

上述语法格式中，在try代码块中编写可能发生异常的Java语句，catch代码块中编写针对异常进行处理的代码。当try代码块中的程序发生了异常，系统会将异常的信息封装成一个异常对象，并将这个对象传递给catch代码块进行处理。catch代码块需要一个参数指明它所能够接收的异常类型，这个参数的类型必须是Exception类或其子类。

4.7.2 try...catch和finally

接下来通过一个案例演示用try…catch语句对异常进行捕获。

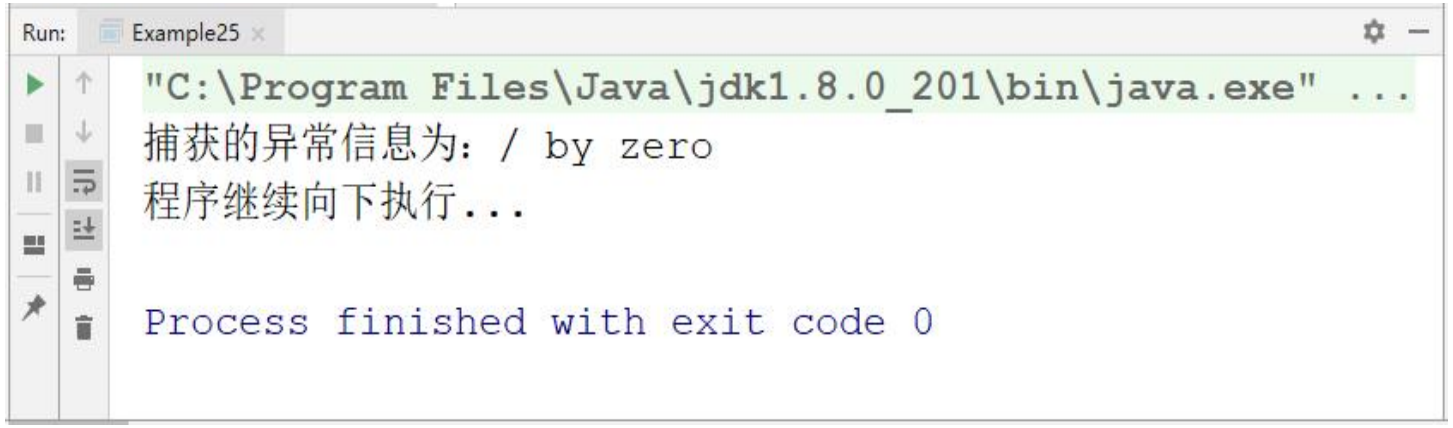
```
1 public class Example25 {
2         public static void main(String[] args) {
3     //下面的代码定义了一个try…catch语句用于捕获异常
4                 try {
5                         int result = divide(4, 0);    //调用divide()方法
6                         System.out.println(result);
7                 } catch (Exception e) {           //对异常进行处理
8                         System.out.println("捕获的异常信息为：" + e.getMessage());
9                 }
10                System.out.println("程序继续向下执行...");
11        }
```

4.7.2 try...catch和finally

```
12 //下面的方法实现了两个整数相除
13         public static int divide(int x, int y) {
14                 int result = x / y;          //定义一个变量result记录两个数相除的结果
15                 return result;               //将结果返回
16         }
17 }
```

4.7.2 try...catch和finally

程序运行结果如下图。



4.7.2 try...catch和finally

上述代码中，第4~9行代码是对可能发生异常的代码用try...catch语句进行了处理。在try代码块中发生除0异常时，程序会通过catch语句捕获异常，第8行代码在catch语句中通过调用Exception对象的getMessage()方法，返回异常信息“/ by zero”。catch代码块对异常处理完毕后，程序仍会向下执行，而不会终止程序。

需要注意的是，在try代码块中，发生异常语句后面的代码是不会被执行的，如上述代码中第6行代码的打印语句就没有执行。

4.7.2 try…catch和finally

在程序中，有时候会希望有些语句无论程序是否发生异常都要执行，这时就可以在try…catch语句后，加一个finally代码块。

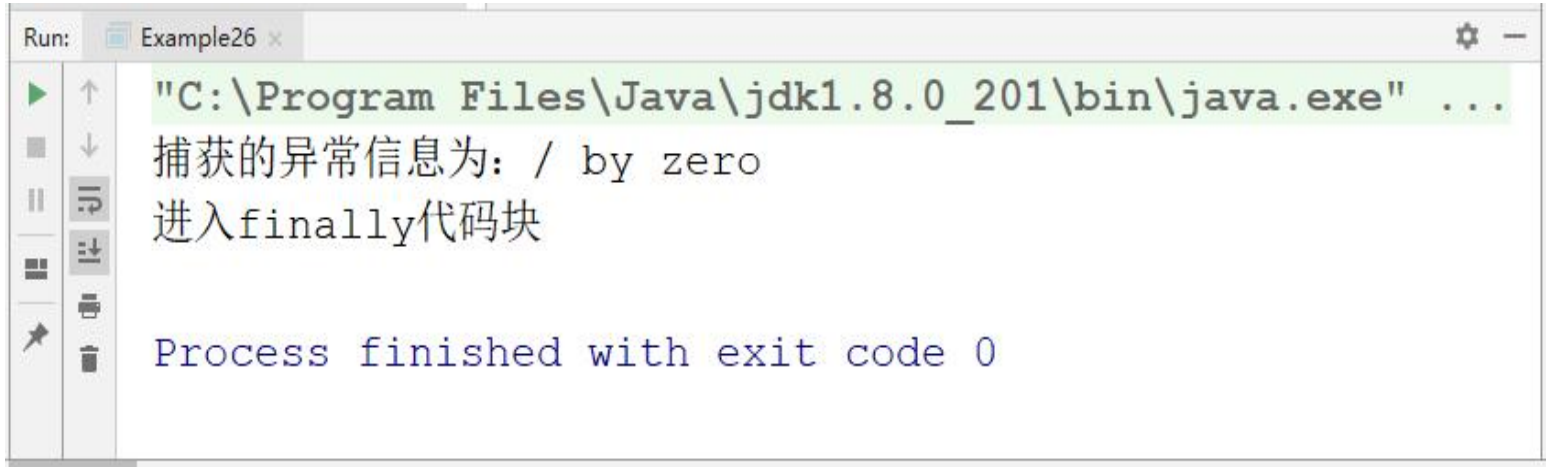
```
1 public class Example26 {
2             public static void main(String[] args) {
3         //下面的代码定义了一个try…catch…finally语句用于捕获异常
4                         try {
5                                 int result = divide(4, 0);        //调用divide()方法
6                                 System.out.println(result);
7                         } catch (Exception e) {                 //对捕获到的异常进行处理
8                                 System.out.println("捕获的异常信息为：" + e.getMessage());
9                                  return;                                 //用于结束当前语句
10                        } finally {
11                                System.out.println("进入finally代码块");
12                        }
```

4.7.2 try...catch和finally

```
13 System.out.println("程序继续向下执行...");
14         }
15     //下面的方法实现了两个整数相除
16         public static int divide(int x, int y) {
17                 int result = x / y;          //定义一个变量result记录两个数相除的结果
18                 return result;               //将结果返回
19         }
20 }
```

4.7.2 try...catch和finally

程序运行结果如下图。



4.7.2 try...catch和finally

上述代码中，第9行代码是在catch代码块中增加了一个return语句，用于结束当前方法，程序第13行代码就不会执行了，而finally代码块中的代码仍会执行，不受return语句影响。也就是说不论程序是发生异常还是使用return语句结束，finally中的语句都会执行。因此，在程序设计时，通常会使用finally代码块处理完成必须做的事情，如释放系统资源。

4.7.2 try...catch和finally

finally中的代码块在一种情况下是不会执行的，那就是在try...catch中执行了System.exit(0)语句。System.exit(0)表示退出当前的Java虚拟机，Java虚拟机停止了，任何代码都不能再执行了。

4.7.3 throws关键字

在实际开发中，大部分情况下我们会调用别人编写方法，并不知道别人编写的方法是否会发生异常。针对这种情况，Java允许在方法的后面使用throws关键字对外声明该方法有可能发生的异常，这样调用者在调用方法时，就明确地知道该方法有异常，并且必须在程序中对异常进行处理，否则编译无法通过。

4.7.3 throws关键字

throws关键字声明抛出异常的语法格式如下：

```
修饰符 返回值类型 方法名(参数1，参数2.....)throws 异常类1，异常类2.....{
        //方法体.....
}
```

从上述语法格式中可以看出，throws关键字需要写在方法声明的后面，throws后面需要声明方法中发生异常的类型。

4.7.3 throws关键字

接下来通过一个案例演示throws关键字的用法。

```
1 public class Example27 {
2         public static void main(String[] args) {
3                 int result = divide(4, 2);     //调用divide()方法
4                 System.out.println(result);
5         }
6     //下面的方法实现了两个整数相除，并使用throws关键字声明抛出异常
7         public static int divide(int x, int y) throws Exception {
8                 int result = x / y;           //定义一个变量result记录两个数相除的结果
9                 return result;                //将结果返回
10        }
11 }
```

4.7.3 throws关键字

编译程序，编译器报错，如下图。

4.7.3 throws关键字

在上述代码中，第3行代码调用divide()方法时传入的第二个参数为2，程序在运行时不会发生被0除的异常，但是由于定义divide()方法时声明了抛出异常，调用者在调用divide()方法时就必须进行处理，否则就会发生编译错误。

4.7.3 throws关键字

接下来修改上述程序，在try…catch处理divide()方法抛出的异常。

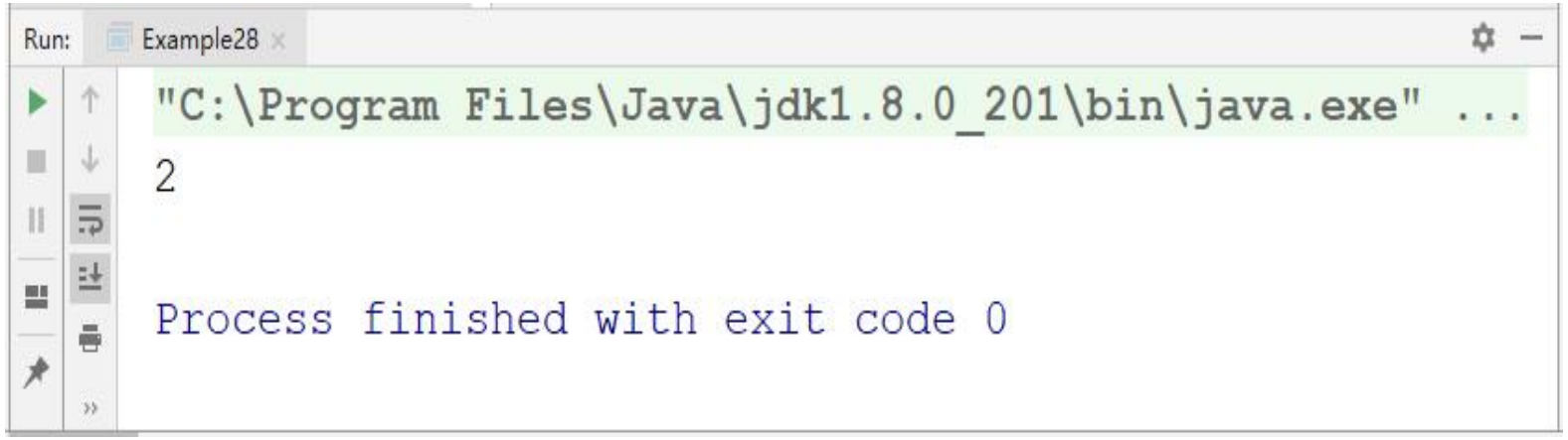
```
1 public class Example28 {
2         public static void main(String[] args) {
3       //下面的代码定义了一个try…catch语句用于捕获异常
4                 try {
5                         int result = divide(4, 2);   //调用divide()方法
7                         System.out.println(result);
8                 } catch (Exception e) {          //对捕获到的异常进行处理
9                         e.printStackTrace();          //打印捕获的异常信息
10                }
11        }
```

4.7.3 throws关键字

```
12 //下面的方法实现了两个整数相除，并使用throws关键字声明抛出异常
13         public static int divide(int x, int y) throws Exception {
14                 int result = x / y;      //定义一个变量result记录两个数相除的结果
15                 return result;           //将结果返回
16         }
17 }
```

4.7.3 throws关键字

上述代码由于使用了try…catch对divide()方法进行了异常处理，因此程序可以编译通过，运行后正确的打印出了运行结果2。程序运行结果如下图。



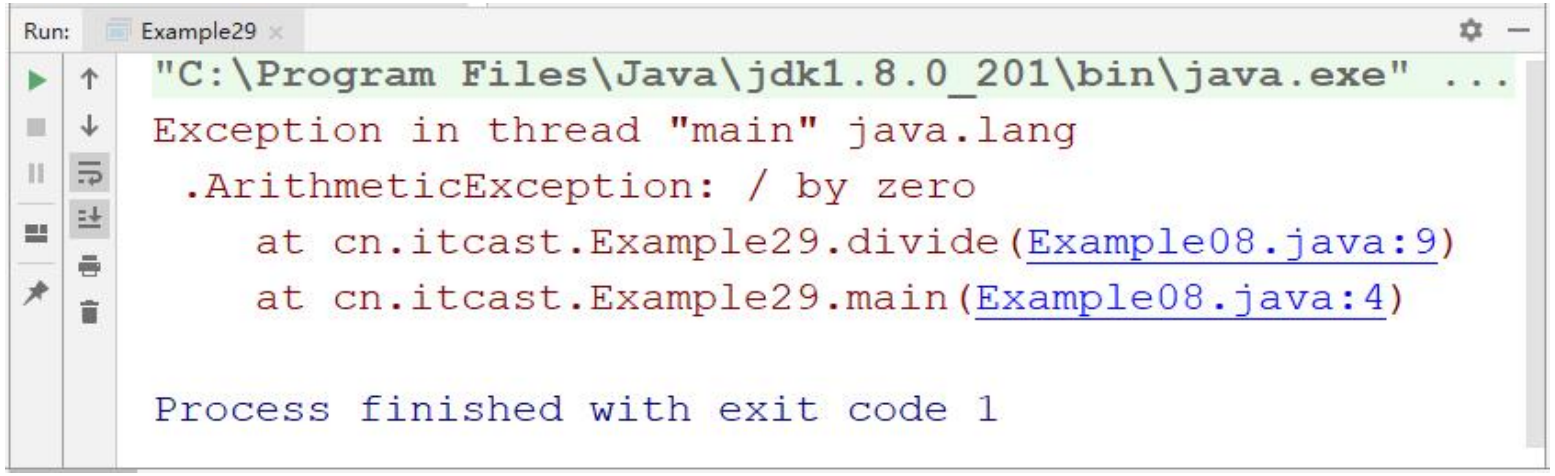
4.7.3 throws关键字

由于使用了try…catch对divide()方法进行了异常处理，因此程序可以编译通过，运行后正确的打印出了运行结果2。下面修改上述程序，将divide()方法抛出的异常继续抛出。

```
1 public class Example29 {
2         public static void main(String[] args)throws Exception {
3                 int result = divide(4, 0);   // 调用divide()方法
4                 System.out.println(result);
5         }
6   // 下面的方法实现了两个整数相除，并使用throws关键字声明抛出异常
7         public static int divide(int x, int y) throws Exception {
8                 int result = x / y;          // 定义一个变量result记录两个数相除的结果
9                 return result;               // 将结果返回
10        }
11 }
```

4.7.3 throws关键字

程序运行结果如下图。



4.7.3 throws关键字

上述代码，在main()方法继续使用throws关键字将Exception抛出，程序虽然可以通过编译，但从运行结果可以看出，在运行时期由于没有对“/by zero”的异常进行处理，最终导致程序终止运行。

4.7.4 运行时异常与编译时异常

在实际开发中，经常会在程序编译时产生一些异常，这些异常必须要进行处理，这种异常被称为编译时异常，也称为checked异常。另外还有一种异常是在程序运行时产生的，这种异常即使不编写异常处理代码，依然可以通过编译，因此被称为运行时异常，也称为unchecked异常。

4.7.4 运行时异常与编译时异常

1．编译时异常

在Exception类中，除了RuntimeException类及其子类，Exception的其他子类都是编译时异常。编译时异常的特点是Java编译器会对异常进行检查，如果出现异常就必须对异常进行处理，否则程序无法通过编译。

处理编译时期的异常有两种方式，具体如下：

（1）使用try…catch语句对异常进行捕获处理。

（2）使用throws关键字声明抛出异常，调用者对异常进行处理。

4.7.4 运行时异常与编译时异常

2．运行时异常

RuntimeException类及其子类都是运行时异常。运行时异常的特点是Java编译器不会对异常进行检查。也就是说，当程序中出现这类异常时，即使没有使用try..catch语句捕获或使用throws关键字声明抛出，程序也能编译通过。运行时异常一般是由程序中的逻辑错误引起的，在程序运行时无法恢复。

4.7.4 运行时异常与编译时异常

例如，通过数组的角标访问数组的元素时，如果角标超过了数组范围，就会发生运行时异常，代码如下所示：

```
int[] arr=new int[5];
System.out.println(arr[6]);
```

在上面的代码中，由于数组arr的length为5，最大角标应为4，当使用arr[6]访问数组中的元素就会发生数组角标越界的异常。

4.7.5 自定义异常

JDK中定义了大量的异常类，虽然这些异常类可以描述编程时出现的大部分异常情况，但是在程序开发中有时可能需要描述程序中特有的异常情况，例如，前面讲解的程序中的divide()方法，不允许被除数为负数。为了解决这个问题，Java允许用户自定义异常，但自定义的异常类必须继承自Exception或其子类。

4.7.5 自定义异常

自定义异常的具体代码如下所示。

```
// 下面的代码是自定义一个异常类继承自Exception
public class DivideByMinusException extends Exception{
        public DivideByMinusException (){
                super();              // 调用Exception无参的构造方法
        }
        public DivideByMinusException (String message){
                super(message);              // 调用Exception有参的构造方法
        }
}
```

4.7.5 自定义异常

在实际开发中，如果没有特殊的要求，自定义的异常类只需继承Exception类，在构造方法中使用super()语句调用Exception的构造方法即可。

自定义异常类中使用throw关键字在方法中声明异常的实例对象，格式如下：

throw Exception异常对象

4.7.5 自定义异常

下面通过一个案例演示自定义异常类。

```
1 public class Example30 {
2         public static void main(String[] args) {
3                 int result = divide(4, -2);
4       System.out.println(result);
5         }
6         //下面的方法实现了两个整数相除
7         public static int divide(int x, int y) {
8                 if(y<0){
9                         throw new DivideByMinusException("除数是负数");                    }
10                int result = x / y;      // 定义一个变量result记录两个数相除的结果
11                return result;           // 将结果返回
12        }
13 }
```

4.7.5 自定义异常

编译程序，编译器报错，如下图。

4.7.5 自定义异常

从运行结果可以看出，程序在编译时就发生了异常。因为在一个方法内使用throw关键字抛出异常对象时，需要使用try…catch语句对抛出的异常进行处理，或者在divide()方法上使用throws关键字声明抛出异常，由该方法的调用者负责处理。但是程序没有这样做。

为了解决上面的问题，对程序进行修改，在divide()方法上，使用throws关键字声明抛出DivideByMinusException异常，并在调用divide()方法时使用try…catch语句对异常进行处理。

4.7.5 自定义异常

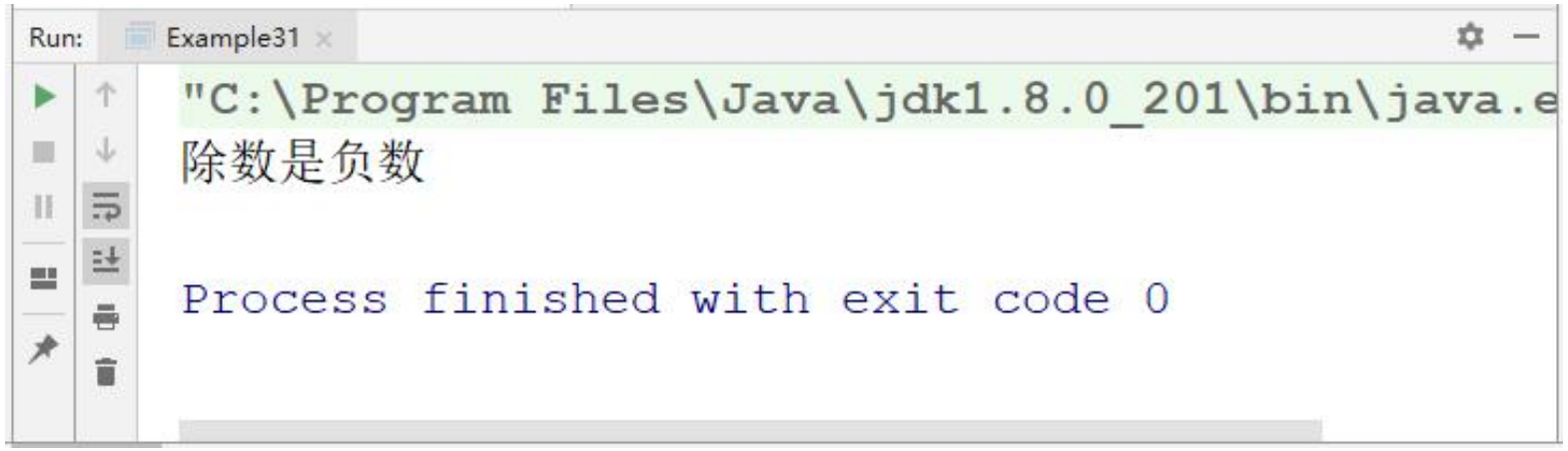
```
1 public class Example31 {
2         public static void main(String[] args) {
3           // 下面的代码定义了一个try…catch语句用于捕获异常
4                   try {
5                           int result = divide(4, -2);
6           System.out.println(result);
7                   } catch (DivideByMinusException e) {     // 对捕获到的异常进行处理
8                           System.out.println(e.getMessage()); // 打印捕获的异常信息
9                   }
10        }
```

4.7.5 自定义异常

```
11 // 下面的方法实现了两个整数相除，并使用throws关键字声明抛出自定义异常
12         public static int divide(int x, int y) throws DivideByMinusException {
13                 if (y < 0) {
14                         throw new DivideByMinusException("除数是负数");
15                 }
16                 int result = x / y;      // 定义一个变量result记录两个数相除的结果
17                 return result;           // 将结果返回
18         }
19 }
```

4.7.5 自定义异常

程序运行结果如下图。



4.7.5 自定义异常

上述代码中的main()方法中，第4~9行代码使用try…catch语句捕获处理divide()方法抛出的异常。在调用divide()方法时，如果传入的被除数不能为负数，程序会抛出一个自定义的DivideByMinusException异常，该异常最终被catch代码块捕获处理，并打印出异常信息。

4.8 本章小结

本章主要介绍了面向对象的继承、多态特性，与第3章学习的面向对象的封装性构成了面向对象语言程序设计的三大特性，这是Java语言的精髓所在。此外，本章还介绍了final关键字、抽象类和接口、Object类、内部类、异常的概念以及异常的处理机制等。本章和第3章，是本书最重要的两章，熟练掌握这两章内容，才是掌握了Java语言的精髓。