第3章 面向对象

Java 的是一种面向对象的编程语言，即“万物皆为对象”。面向对象的思想是最为简单的编程思想，也最接近人类的思维习惯，本章将为大家详细讲解关于面向对象的的编程知识。

3.1 面向对象的概念和特征

现实生活中存在各种形态不同的事物，这些事物之间存在着各种各样的联系。在程序中使用对象来映射现实中的事物，使用对象的关系来描述事物之间的联系，这种思想就是面向对象。面向对象的特征主要可以概括为封装性、继承性和多态性，接下来针对这三种特征进行简单介绍。

1．封装性

封装是面向对象的核心思想，将对象的属性和行为封装了起来，不需要让外界知道内部是如何实现细节的，这就是封装的思想。例如，使用电视机的用户不需要了解电视机内部复杂工作的具体细节，他们只需要知道开、关、选台、调台等这些设置与操作就可以了。

2．继承性

继承性是描述类与类之间的关系，在已有类的基础上扩展出新的类。例如，有一个火车类，该类描述了火车的特性和功能，而高铁类中不仅应该包含火车的特性和功能，还应该增加高铁特有的功能，这时可以让高铁类继承火车类，在高铁类中单独添加高铁特有的方法就可以了。继承不仅增强了代码的复用性，提高了开发效率，同时还为后期的代码维护提供了便利。

3．多态性

多态性指的是对象在不同情况下具有不同的表现能力。在一个类中定义的属性和方法被其他的类继承后，它们可以表现出不同的行为，使得同一个属性和方法在不同的类中具有不同的意义。

3.2 类与对象

3.2.1 类与对象的关系

面向对象的编程思想是让程序代码中对事物的描述和在现实中事物的形态相关联。为了实现这些，在面向对象的思想中提出了两个概念，即类和对象。其中，类是一组具有共同特征和行为的对象的抽象描述，而对象是表示该类事物的具体个体。接下来通过一个图例来抽象描述类与对象的关系，如图 3-1 所示。

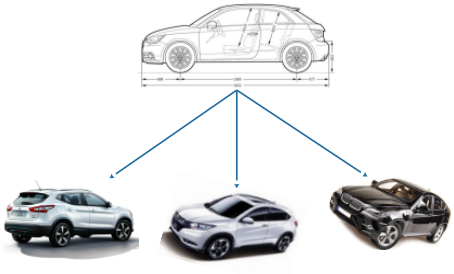

图3-1 类与对象

在图 3-1 中可以看出，汽车图纸就相当于一个类，每个生成出来的汽车就相当于一个对象。因为汽车

本身属于一种广义的概念，并不能代表具体的东西。从汽车类到具体的某个汽车便可以看出类用于描述多个对象的共同特征，它是对象的模板。而对象是用于描述现实中的个体，它是类的实例。

3.2.2 类的定义

对象是面向对象思想中的核心，为了在程序中创建对象，首先需要定义一个类。类是通过“Class”关键字来定义的，类中可以定义成员变量和成员方法，其中成员变量用于描述对象的特征（也称作属性），成员方法用于描述对象的行为（简称为方法）。

假设要在程序中描述汽车的相关信息，可以先设计一个汽车类，在这个类中定义两个属性 color、num 分别表示车的颜色和轮胎，定义一个方法 run()表示车跑的行为。接下来根据这个描述来设计一个 Car 类，首先创建一个 chapter03 项目，然后在该项目下创建一个 com.itheima.example01 包，在该包下创建一个 Car 类，如文件 3-1 所示。

文件3-1 Car.java

```
1  package com.itheima.example01;
2  public class Car {
3      String color;   //定义一个颜色属性
4      int num;        //定义一个轮胎属性
5      public void run() {
6          // 方法中打印属性 color 和 num 的值
7          System.out.println("这辆车的颜色是" + color + ",轮胎数量是" + num);
8      }
9   }
```

在文件 3-1 中，定义了一个 Car 类。其中，Car 是类名，color 和 num 是成员变量，run ()是成员方法。在成员方法 run ()中可以直接访问成员变量 color 和 num。

3.2.3 对象的创建

前一小节提到过对象是面向对象的核心，因此仅有类是不够的，还需要创建对象，通过对象的引用来访问类中的成员。在 Java 中通过 new 关键字来创建对象，具体格式如下：

```
类名 对象名 = new 类名();
```

例如，创建 Car 类的实例对象代码如下：

```
Car c = new Car();
```

在上述代码中，通过“new”关键字创建了一个 Car 的实例对象，“Car c”则是声明了一个 Car 类型的变量 c。中间的等号用于将 Car 对象在内存中的地址赋值给变量 c，这样变量 c 便持有了对象的引用。在创建 Car 对象后，便可以通过“对象的引用.对象成员”的方式来访问该对象所有的成员，接下来在当前包中创建一个 Example01 类来访问对象的成员，如文件 3-2 所示。

文件3-2 Example01.java

```
1  package com.itheima.example01;
2  class Example01 {
3      public static void main(String[] args) {
4          Car c1 = new Car(); // 创建第一个 Car 对象
5          Car c2 = new Car(); // 创建第二个 Car 对象
6          c1.color = "red";    // 为 color 属性赋值
```

```
7           c1.num = 4;
8           c1.run(); // 调用对象的方法
9           c2.run();
10      }
11  }
```

运行结果如图 3-2 所示。

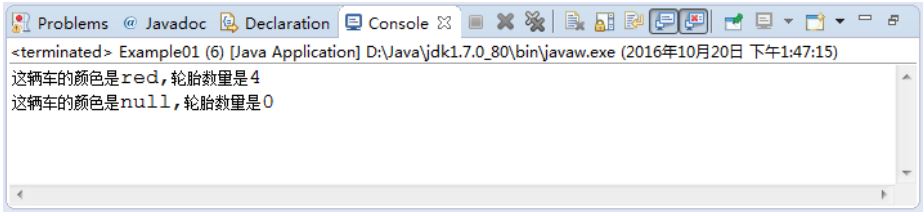

图3-2　运行结果

从图 3-2 所显示的运行结果可以看出，c2 对象的 color 属性也是有值的，其默认值为 null。这是因为在实例化对象时，JVM 会自动为成员变量进行初始化，针对不同数据类型的成员变量，JVM 会赋予不同的默认值，如 long 类型默认初始值为 0L，boolean 默认初始值为 false 等。

3.2.4 类的封装

类的封装是指，在定义一个类时将类中的属性进行私有化，即使用 private 关键字进行修饰，私有化属性只能在当前所在类中被访问到，如果外界想要访问私有属性，需要提供一些 public 修饰的公共方法，其中包括用于获取属性值的 getter 方法和设置属性值的 setter 方法。

接下来在 src 目录下创建一个 com.itheima.example02 包（由于一个包中不能出现同名的类，因此需要再创建一个包），在该包下创建一个 Car 类，来实现类的封装，如文件 3-3 所示。

文件3-3　Car.java

```
1  package com.itheima.example02;
2  public class Car {
3      private String color;   // 将 color 属性私有化
4      private int num;        // 将 num 属性私有化
5      // 下面是公有的 getter 和 setter 方法
6      public String getColor() {
7          return color;
8      }
9      public void setColor(String color) {
10         this.color = color;  //this.color 访问的是成员变量，后面会详细讲解
11     }
12     public int getNum() {
13         return num;
14     }
15     public void setNum(int carNum) {
16         //对传入的参数进行检查
17         if (carNum != 4) {
18             System.out.println("输入的轮胎数量不正确！");
19         } else {
```

```
20              num = carNum;   // 给属性赋值
21          }
22      }
23      public void run() {
24          System.out.println("这辆车的颜色是" + color + ",轮胎数量是" + num);
25      }
26  }
```

在文件 3-3 中，Car 类定义了两个私有化的成员变量 color 和 num，并为这两个属性提供了公共的 getter 和 setter 方法，其中 getColor()和 getNum()方法用于获取属性的值，而 setColor ()和 setNum ()方法用于设置属性的值。

接下来创建测试类 Example02，在该类中创建一个 Car 对象，然后调用了 setNum()、setColor()以及 run()方法，如文件 3-4 所示。

文件3-4 Example02.java

```
1  package com.itheima.example02;
2  public class Example02 {
3      public static void main(String[] args) {
4          Car c = new Car();
5          c.setNum(-1);
6          c.setColor("red");
7          c.run();
8      }
9  }
```

运行结果如图 3-3 所示。

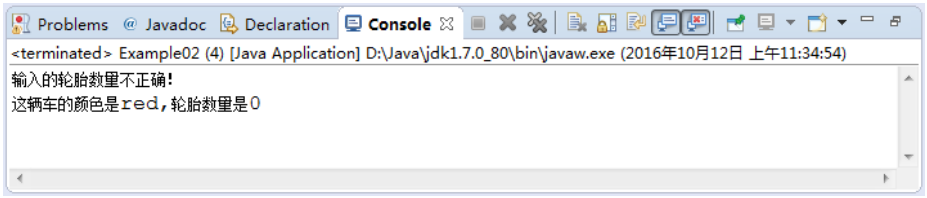

图3-3 运行结果

从图 3-3 的运行结果可以看出，在 setNum()方法中对参数 carNum 进行了校验，由于-1 不符合规则，因此在控制台打印为“输入的轮胎数量不正确!”，num 属性没有被赋值，仍为初始值 0。

3.3 构造方法

3.3.1 构造方法的定义

在实例化对象赋值时，不仅可以通过 setter 方法来完成，还可以通构造方法来完成。所谓的构造方法是类的一个特殊成员，它会在类实例化对象时被自动调用。在定义构造方法时，必须要同时满足三个条件，具体如下：

① 方法的名称和类名必须相同；

② 在方法名称的前没有返回值类型的声明；

③ 在方法体中不可以使用 return 语句返回一个值，但允许单独写 return 语句来作为方法的结束。

接下来通过一个案例来演示如何在类中定义构造方法，如文件 3-5 所示。

文件3-5 Car.java

```
1  package com.itheima.example03;
2  public class Car {
3      // 下面是类的构造方法
4      public Car() {
5          System.out.println("无参数的构造方法执行了...");
6      }
7  }
```

在文件3-5中，Car类只定义了一个无参的构造方法，用于对象的初始化。接下来创建测试类Example03，调用无参构造方法，如文件3-6所示。

文件3-6 Example03.java

```
1  package com.itheima.example03;
2  public class Example03 {
3      public static void main(String[] args) {
4          Car c = new Car();  // 实例化 Car 对象
5      }
6  }
```

运行结果如图3-4所示。

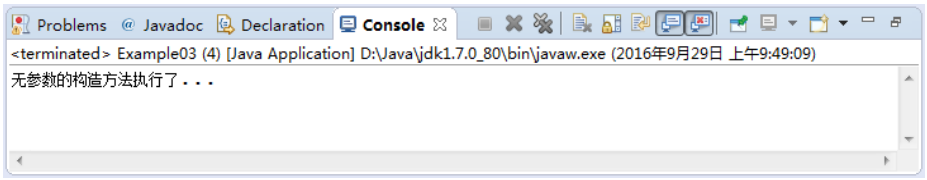

图3-4 运行结果

从图3-4的运行结果可以看出，Car类的无参数的构造方法执行了，这是因为在实例化对象时会自动的调用该类的构造方法。

在类中既然可以定义无参的构造方法，那么是否可以定义有参的构造方法并通过参数对属性赋值呢？接下来通过一个案例进行演示，如文件3-7所示。

文件3-7 Car.java

```
1  package com.itheima.example04;
2  public class Car {
3      String color;
4      // 定义有参的构造方法
5      public Car (String c) {
6          color = c;            // 为 color 属性赋值
7      }
8      public void run() {
9          System.out.println("这辆车的颜色是" + color);
10     }
11 }
```

在文件3-7中，定义了一个成员变量和一个有参的构造方法，该方法执行时，会自动为其成员变量进行赋值。接下来创建测试类Example04，调用有参构造方法，如文件3-8所示。

文件3-8 Example04.java

```
1  package com.itheima.example04;
```

```
2  public class Example04 {
3      public static void main(String[] args) {
4          Car c = new Car("red"); // 实例化 Car 对象
5          c.run();
6      }
7  }
```

运行结果如图 3-5 所示。

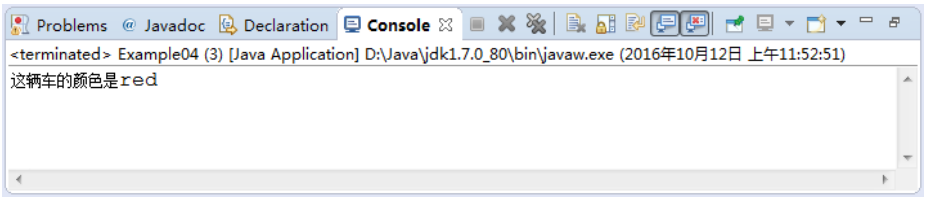

图3-5 运行结果

从图 3-5 的运行结果可以看出，Car 对象在调用 run()方法时，其 color 属性已经被赋值为“red”。

3.3.2 构造方法的重载

构造方法与普通方法一样也可以实现重载，由于构造方法的方法名与类名相同，因此只要每个构造方法的参数类型和参数个数不同即可实现构造方法的重载。在创建实例对象时，可以通过调用不同的构造方法来为不同的属性进行赋值。接下来通过一个案例来学习构造方法的重载，如文件 3-9 所示。

文件3-9 Car.java

```
1  package com.itheima.example05;
2  public class Car {
3      String color;
4      int num;
5      // 定义两个参数的构造方法
6      public Car(String c_color, int c_num) {
7          color = c_color;    // 为 color 属性赋值
8          num = c_num;        // 为 num 属性赋值
9      }
10     // 定义一个参数的构造方法
11     public Car(String c_color) {
12         color = c_color;    // 为 color 属性赋值
13     }
14     public void run() {
15         // 打印 color 和 num 的值
16         System.out.println("这辆车的颜色是" + color + ",轮胎数量是" + num);
17     }
18 }
```

在文件 3-9 中，定义了两个重载的构造方法，在创建对象时，根据传入参数的不同，分别调用不同的构造方法。接下来创建测试类 Example05，分别调用两个构造方法，如文件 3-10 所示。

文件3-10 Example05.java

```
1  package com.itheima.example05;
2  public class Example05 {
```

```
3      public static void main(String[] args) {
4          // 分别创建两个对象 c1 和 c2
5          Car c1 = new Car("red");
6          Car c2 = new Car("black", 4);
7          // 通过对象 c1 和 c2 调用 run()方法
8          c1.run();
9          c2.run();
10     }
11 }
```

运行结果如图 3-6 所示。

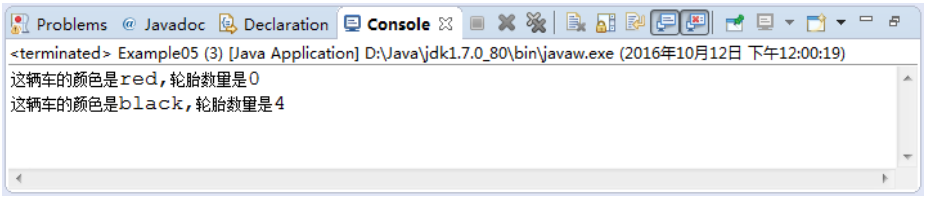

图3-6 运行结果

从图 3-6 的运行结果可以看出，两个构造方法对属性赋值的情况是不一样的，其中一个参数的构造方法只针对 color 属性进行赋值，这时 num 属性的值为默认值 0。

3.4 this 关键字

在 3.3.2 小节的案例中，在成员变量上使用的是 num，在构造方法中使用的是 c_num，这样类似的变量一旦增多，可读性会变得很差，这时需要在一个类中使用统一的变量名称表示年龄。为了解决这样的问题，Java 中提供一个 this 关键字表示当前对象，可以在方法中调用其他的成员。

接下来详细讲解 this 关键字在程序中的三种常见用法，具体如下：

①通过 this 关键字可以明确地去访问一个类的成员变量，解决与局部变量名称相同问题。具体示例代码如下：

```
1  public class Car {
2      String color;
3      public Car(String color) {
4          this.color = color;
5      }
6      public int getColor() {
7          return this.color;
8      }
9  }
```

在上面的代码中，构造方法的参数被定义为 color，它是一个局部变量，在类中还定义了一个成员变量，名称也是 color。在构造方法中如果使用“color”，则是访问局部变量，但如果使用“this.color”则是访问成员变量。

②通过 this 关键字调用成员方法，具体示例代码如下：

```
1  public class Car {
2      public void show() {
3      }
4      public void run() {
```

```
5           this.show();
6       }
7   }
```

在上面的 run()方法中，使用 this 关键字调用 show()方法。注意，此处的 this 关键字可以省略不写，也就是说上面的代码中，写成“this.show()”和“show()”，效果是完全一样的。

构造方法是在实例化对象时被 JVM 自动调用的，在程序中不能像调用其他方法一样去调用构造方法，但可以在一个构造方法中使用“this([参数 1,参数 2…])”的形式来调用其他的构造方法。具体代码如文件 3-11 所示。

文件3-11 Car.java

```
1  package com.itheima.example06;
2  public class Car {
3      public Car() {
4          System.out.println("无参数的构造方法执行了...");
5      }
6      public Car(String color) {
7          this();                  // 调用无参的构造方法
8          System.out.println("有参数的构造方法执行了...");
9      }
10  }
```

在文件 3-11 中，定义了两个重载的构造方法，在上述代码的第 7 行调用了 this()，此方法会调用无参的构造方法。接下来创建测试类 Example06，调用有参的构造方法，如文件 3-12 所示。

文件3-12 Example06.java

```
1  package com.itheima.example06;
2  public class Example06 {
3      public static void main(String[] args) {
4          Car c = new Car("red"); // 实例化 Car 对象
5      }
6  }
```

运行结果如图 3-7 所示。

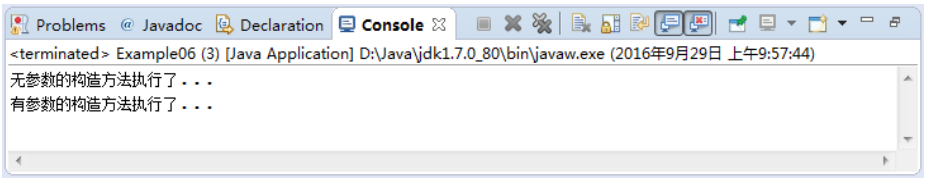

图3-7 运行结果

从图 3-6 的运行结果可以看出，先执行了无参的构造方法，之后再执行有参的构造方法。需要注意的是，使用 this 关键字调用其他构造方法时只能出现在构造方法中，并且只能位于构造方法的第一行且只能出现一次。另外不能在两个构造方法中使用 this 相互调用，否则会出现编译错误。

3.5 static 关键字

3.5.1 静态变量

通过前面的学习读者已经了解，如果使用一个类则会在产生实例化对象时分别在堆内存中分配空间，在堆内存中要保存对象中的属性，每个对象有自己的属性，如果有些属性希望被所有对象共享，就必须使用 static 关键字修饰成员变量，该变量被称作静态变量，可以直接使用“类名.变量名”的形式来调用，如文件 3-13 所示。

文件3-13 Car.java

```
1  package com.itheima.example07;
2  public class Car {
3      static String carName = "大众"; // 定义静态变量 carName
4  }
```

在文件 3-13 中，定义了一个静态变量 carName，用于表示汽车所在的厂商，它被所有的实例所共享。接下来创建测试类 Example07，在该类中分别通过 Car.carName 和 c2.carName 的方式来调用静态变量，如文件 3-14 所示。

文件3-14 Example07.java

```
1  package com.itheima.example07;
2  public class Example07 {
3      public static void main(String[] args) {
4          Car c1 = new Car();
5          Car.carName = "大众"; // 为静态变量赋值
6          System.out.println("该辆车的厂商是：" + Car.carName);
7          System.out.println("该辆车的厂商是：" + c1.carName);
8      }
9  }
```

运行结果如图 3-8 所示。

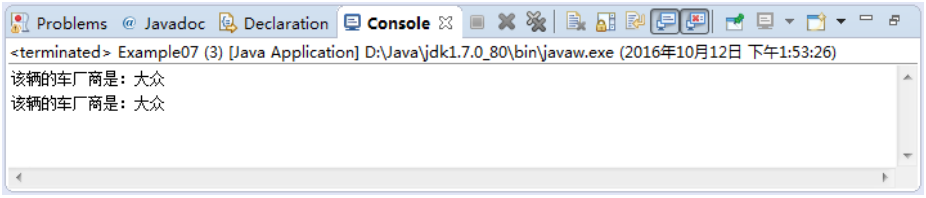

图3-8 运行结果

从图 3-8 的运行结果可以看出，Car.carName 和 c2.carName 两种不同的访问方式，打印结果均为“大众”。

注意：

static 关键字只能用于修饰成员变量，不能用于修饰局部变量，否则编译会报错，下面的代码是非法的。

```
public class Car {
    public void run() {
        static int number = 4; // 这行代码是非法的，编译会报错
    }
```

```
    }
```

3.5.2 静态方法

通过前面讲解可知，要想调用某个方法必须要创建一个对象，那我们的头脑中一定有这样一个疑问，有没有一种方法不创建对象就能直接调用呢？实际上是有一种方法可以这样操作的，那就是静态方法，静态方法与普通方法的区别是在方法前面加一个“static”关键字，这种方法称为静态方法。同静态变量一样，静态方法可以使用“类名.方法名”的方式来访问，也可以通过类的实例对象来访问。接下来通过一个案例来学习静态方法的使用，如文件 3-15 所示。

文件3-15 Car.java

```
1  package com.itheima.example08;
2  public class Car {
3      public static void run() { // 定义静态方法
4          System.out.println("run()方法执行了...");
5      }
6   }
```

在文件 3-15 中可以看出，该类中只定义了一个静态方法 run()。接下来创建测试类 Example08，在该类中分别使用两种方式来调用静态方法，如文件 3-16 所示。

文件3-16 Example08.java

```
1  package com.itheima.example08;
2  public class Example08 {
3      public static void main(String[] args) {
4          // 1.类名.方法的方式调用静态方法
5          Car.run();
6          // 2.实例化对象的方式来调用静态方法
7          Car c = new Car();
8          c.run();
9      }
10 }
```

运行结果如图 3-9 所示。

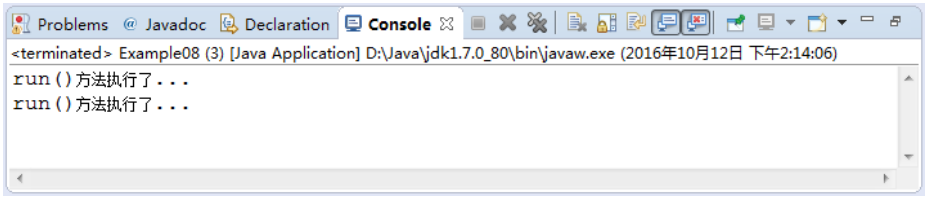

图3-9 运行结果

从图 3-9 的运行结果可以看出，分别使用了两种不同的方式都成功调用了静态方法，这说明通过实例化的对象同样可以调用静态方法。

注意：

在一个静态方法中只能访问用 static 修饰的成员，原因在于没有被 static 修饰的成员需要先创建对象才能访问，而静态方法在被调用时可以不需要创建任何对象。

3.5.3 静态代码块

在 Java 类中，使用一对大括号包围起来的若干行代码被称为一个代码块，用 static 关键字修饰的代码块称为静态代码块。当类被加载时，静态代码块会执行，由于类只加载一次，所以静态代码块只会执行一次。在程序中，通常会使用静态代码块来对类的成员变量进行初始化。接下来通过一个案例来了解静态代码块的使用，如文件 3-17 所示。

文件3-17 Car.java

```
1  package com.itheima.example09;
2  public class Car {
3      static String color;
4      // 下面是一个静态代码块
5      static {
6          color = "red";
7          System.out.println("这辆车的颜色是" + color);
8      }
9  }
```

在文件 3-17 中可以看出，当静态代码块执行时，会为其静态变量赋值为“red”，并打印结果。接下来创建测试类 Example09，在 main()方法中创建两个 Car 的实例对象 c1 和 c2，如文件 3-18 所示。

文件3-18 Example09.java

```
1  package com.itheima.example09;
2  public class Example09 {
3      public static void main(String[] args) {
4          // 下面的代码创建了两个 Car 对象
5          Car c1 = new Car();
6          Car c2 = new Car();
7      }
8  }
```

运行结果如图 3-10 所示。

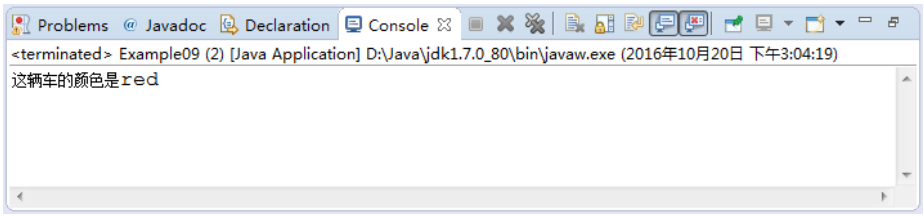

图3-10 运行结果

从图 3-10 的运行结果可以看出，在两次创建对象的过程中，静态代码块中的内容只输出了一次，这就说明静态代码块在类中只会加载一次。

3.6 类的继承

3.6.1 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产生一个新的类，产生的新类被称作子类，现有类被称作父类，子类会自动拥有父类的属性和方法。例如定义一个 Animal 类作为父类，该类拥有一个 call()方法，当子类 Cow 和 Sheep 继承自 Animal 类时，就会自动拥有 call()方法，接下来通过一个图例来描述类的继承关系，如图 3-11 所示。

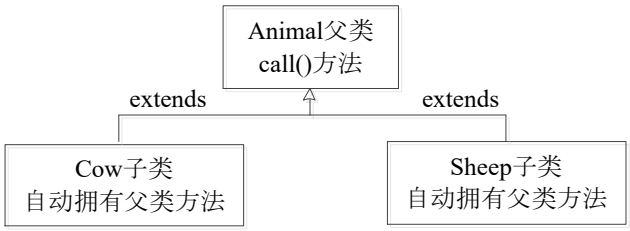

图3-11　动物继承关系图

在程序代码中，如果想定义一个类继承另一个类，需要使用 extends 关键字。如果一个类没有使用 extends 关键字明确标识继承另一个类，那么这个类就是默认继承 Object 类。Object 类是所有类的父类，那么该类中的方法适用于他的所有子类，其类中常用的方法有 toString()、hashCode()方法等。

注意：

在类的继承中，需要注意以下几个问题，具体如下：

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 Cow 这个类继承 Animal 之后，就不允许继承其他类。

2、多个类可以继承一个父类，例如 Cow 和 Sheep 都可以继承 Animal 类。

3、在 Java 中，多层继承是允许的，即一个类的父类可以再去继承其他的父类，例如 Zebra 类继承自 Horse 类，而 Horse 类又可以去继承 Animal 类，即子孙三代。

3.6.2 重写父类方法

在继承关系中，子类会继承父类中定义的方法，但子类也可以在父类的基础上拥有自己的特征，即对父类的方法进行重写。需要注意的是，在子类中重写的方法必须与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。接下来通过一个案例来演示如何实现子类重写父类的方法。具体步骤如下：

1. 创建 Animal 类

创建一个 Animal 类，并在类中定义两个方法 call()和 sleep()，分别在两个方法中输出“动物发出的声音”和“动物在睡觉”的信息，如文件 3-19 所示。

文件3-19　Animal.java

```
1  package com.itheima.example10;
2  // 定义 Animal 类
3  public class Animal {
4      // 定义动物叫的方法
5      public void call() {
```

```
6           System.out.println("动物发出声音...");
7       }
8       // 定义动物睡觉的方法
9       public void sleep() {
10          System.out.println("动物在睡觉...");
11      }
12  }
```

2. 创建 Cow 类

创建一个 Cow 类，使其继承 Animal 类，并在 Animal 类重写 call()方法，如文件 3-20 所示。

文件3-20 Cow.java

```
1  package com.itheima.example10;
2  // 定义 Cow 类继承 Animal 类
3  public class Cow extends Animal {
4      // 定义一个打印 name 的方法
5      public void call() {
6          System.out.println("哞...");
7      }
8  }
```

3. 创建测试类

创建一个测试类 Example10，分别调用 call()方法和 sleep()方法，如文件 3-21 所示。

文件3-21 Example10.java

```
1  package com.itheima.example10;
2  public class Example10 {
3      public static void main(String[] args) {
4          Cow c = new Cow();    // 创建一个 Cow 类的实例对象
5          c.call();             // 调用 Cow 重写的 call()方法
6          c.sleep();            // 调用父类 Animal 的 sleep()方法
7      }
8  }
```

运行结果如图 3-12 所示。

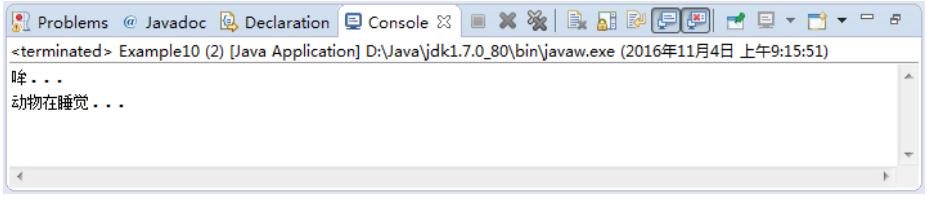

图3-12 运行结果

从图 3-12 的运行结果可以看出，当子类继承父类的时候，会拥有父类中所有的成员，由此可知，在调用 sleep()方法时，会自动调用父类的 sleep()方法。当子类重写了父类方法之后，创建的子类对象会自动调用重写的方法，即调用的是子类的 call()方法。

注意：

子类重写父类方法时，不可以使用比父类中被重写的方法更严格的访问权限，如：父类中的方法是 public 的，子类的方法就不可以是 private 的。

3.6.3 super 关键字

从文件 3-12 的运行结果可以看出，当子类重写父类的方法后，子类对象将不能访问父类被重写的方法，为了解决这个问题，在 Java 中专门提供了 super 关键字用于访问父类的成员变量、成员方法和构造方法，具体格式如下：

```
super.成员变量    //访问成员变量
super.成员方法([参数 1,参数 2…]) //访问成员方法
super([参数 1,参数 2…])          //访问构造方法
```

接下来就通过一个案例来学习 super 的使用，具体步骤如下：

1. 创建 Animal 类

创建一个 Animal 类，并在类中定义一个有参的构造方法，并在该方法中输出一句话，如文件 3-22 所示。

文件3-22 Animal.java

```
1  package com.itheima.example11;
2  // 定义 Animal 类
3  public class Animal {
4      // 定义 Animal 类有参的构造方法
5      public Animal(String name) {
6          System.out.println("我是一头" + name);
7      }
8  }
```

2. 创建 Cow 类

创建一个 Cow 类，使其继承 Animal 类，并定义一个无参的构造方法，该方法中使用 super 关键字，调用父类有参的构造方法，如文件 3-23 所示。

文件3-23 Cow.java

```
1  package com.itheima.example11;
2  // 定义 Cow 类继承 Animal 类
3  class Cow extends Animal {
4      public Cow() {
5          super("黄牛"); // 调用父类有参的构造方法
6      }
7  }
```

3. 创建测试类

创建测试类 Example11，在 main()方法中实例化子类对象，如文件 3-24 所示。

文件3-24 Example11.java

```
1  package com.itheima.example11;
2  // 定义测试类
3  public class Example11 {
4      public static void main(String[] args) {
5          Cow c = new Cow(); // 实例化子类 Cow 对象
6      }
7  }
```

运行结果如图 3-14 所示。

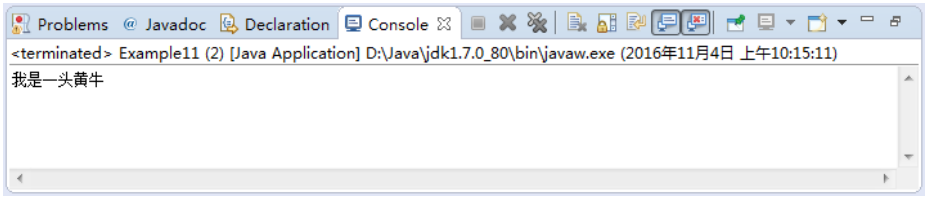

图3-14 运行结果

从图 3-14 的运行结果可以看出，Cow 类的构造方法被调用时父类的构造方法也被调用了。需要注意的是，通过 super 关键字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。

3.7 final 关键字

在 Java 中，如果父类的某些方法不希望再被子类重写，则必须把它们修饰为最终方法，即使用 final 修饰。final 关键字可用于修饰类、变量和方法，表示不能改变或者最终，因此被 final 修饰的类、变量和方法将具有以下特性：

- final 修饰的类不能被继承；
- final 修饰的方法不能被子类重写；
- final 修饰的变量（成员变量和局部变量）是常量，并且只能赋值一次。

接下来通过具体示例来演示 final 关键字的三种特性。

1. Java 中，使用 final 关键字修饰的类不能有子类，具体示例代码如下所示：

```
final class A {}     // 使用 final 修饰类，不能被继承
class B extends A{} // 编译失败，不能继承被 final 修饰的类
```

2. Java 中，使用 final 关键字修饰的方法不能被子类覆盖，具体示例代码如下所示：

```
class A {
    public final void print(){}   // 使用 final 关键字声明的方法不能被重写
}
class B extends A {
    public final void print(){}   // 编译失败，不能重写用 final 修饰的方法
}
```

3. Java 中，被 final 修饰的变量即变为常量，常量是不能被修改的，具体示例代码如下所示：

```
class A {
    final String NAME = "XXX";    // 使用 final 声明的变量就是常量
    public void print(){
        NAME = "YYY";             // 编译失败，常量不可修改
    }
}
```

3.8 抽象类和接口

3.8.1 抽象类

在 Java 程序中，允许在定义方法时不写方法体，这种方法被称为抽象方法，抽象方法必须使用 abstract

关键字修饰。抽象方法的出现解决了程序中某些方法的不确定实现，例如前面在定义 Animal 类时，call()方法用于表示动物的叫声，但是针对不同的动物，叫声也是不同的，因此在 call()方法中无法准确描述动物的叫声。抽象方法的语法格式如下：

```
abstract void call();      // 定义抽象方法 call()
```

如果一个类中定义了抽象方法，则该类必须定义为抽象类，抽象类也同样使用 abstract 关键字来修饰，具体示例如下：

```
// 定义抽象类 Animal
abstract class Animal {
    abstract int call();  // 定义抽象方法 call()
}
```

需要注意的是，包含抽象方法的类必须声明为抽象类，但抽象类可以不包含任何抽象方法，只需使用 abstract 关键字来修饰即可。另外，抽象类是不可以被实例化的，因为抽象类中有可能包含抽象方法，抽象方法是没有方法体的，不可以被调用。如果想调用抽象类中定义的方法，则需要创建一个子类，在子类中将抽象类中的抽象方法进行实现。

接下来通过一个案例来学习如何实现抽象类中的方法，具体步骤如下：

1. 创建 Animal 类

创建一个 Animal 抽象类，并在类中定义一个抽象 call()方法，如文件 3-25 所示。

文件3-25 Animal.java

```
1  package com.itheima.example12;
2  // 定义抽象类 Animal
3  public abstract class Animal {
4      abstract void call();  // 定义抽象方法 call()
5  }
```

2. 创建 Cow 类

创建一个 Cow 类，使其继承 Animal 抽象类，并在 Animal 类重写抽象方法 call()，如文件 3-26 所示。

文件3-26 Cow.java

```
1  package com.itheima.example12;
2  // 定义 Cow 类继承抽象类 Animal
3  public class Cow extends Animal {
4      // 实现抽象方法 call()
5      void call() {
6          System.out.println("哞...");
7      }
8  }
```

3. 创建测试类

创建一个测试类 Example12，在 main()方法中创建了一个子类 Cow 对象，并调用该对象的 call()方法，如文件 3-27 所示。

文件3-27 Example12.java

```
1  package com.itheima.example12;
2  // 定义测试类
3  public class Example12 {
4      public static void main(String[] args) {
5          Cow c = new Cow(); // 创建 Cow 类的实例对象
```

```
6           c.call();           // 调用 cow 对象的 call()方法
7       }
8   }
```

运行结果如图 3-15 所示。

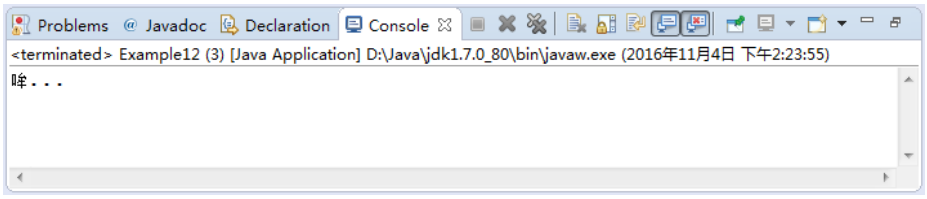

图3-15　运行结果

从图 3-15 的运行结果可以看出，子类实现了父类的抽象方法后，可以正常进行实例化，并通过实例化对象调用子类中的方法。

3.8.2 接口

在 Java 中还存在一种特殊的抽象类，该类中只存在常量和抽象方法，而不存在变量的定义和方法实现，这种特殊的类被称之为接口。接口是由常量和抽象方法组成的特殊类，是对抽象类的进一步抽象。定义接口时，需要使用 interface 关键字来修饰，具体示例代码如下：

```
interface Animal{
    String ANIMAL_ACTION ="动物的行为动作" ;
     void call();
}
```

在上述代码中可以看出，接口中定义了一个全局常量和一个抽象方法，全局常量默认使用了“public static final”来修饰，抽象方法默认使用了“public abstract”来修饰。需要注意的是在接口定义方法时，所有的方法必须都是抽象的，所以不能通过实例化对象的方式调用接口中的方法。此时需要定义一个类，并使用 implements 关键字实现接口中所有的方法。接口的实现类的示例代码具体如下：

```
class Cow implements Animal{
    public void call(){…}
}
```

接下来通过一个案例来学习接口的使用，具体步骤如下：

1. 创建 Animal 接口

创建一个 Animal 接口，并在接口中定义一个全局常量和抽象方法 call()，如文件 3-28 所示。

文件3-28　Animal.java

```
1  package com.itheima.example13;
2  // 定义了 Animal 接口
3  interface Animal {
4      String ANIMAL_ACTION = "动物的行为动作";  // 全局常量,其默认修饰为 public static final
5      void call();     // 抽象方法 call(),其默认修饰为 public abstract
6  }
```

2. 创建 Cow 类

创建一个 Cow 类，使其实现 Animal 接口，并在 Animal 接口实现抽象方法 call()，如文件 3-29 所示。

文件3-29　Cow.java

```
1  package com.itheima.example13;
```

```
// Cow 类实现了 Animal 接口
class Cow implements Animal {
    // 实现 call()方法
    public void call() {
        System.out.println(ANIMAL_ACTION+":"+"哞...");
    }
}
```

3. 创建测试类

创建一个测试类 Example13，在 main()方法中创建了一个子类 Cow 对象，并该对象的 call()方法，如文件 3-30 所示。

文件3-30 Example13.java

```
package com.itheima.example13;
// 定义测试类
public class Example13 {
    public static void main(String args[]) {
        Cow c = new Cow(); // 创建 Cow 类的实例对象
        c.call();          // 调用 cow 对象的 call()方法
    }
}
```

运行结果如图 3-16 所示。

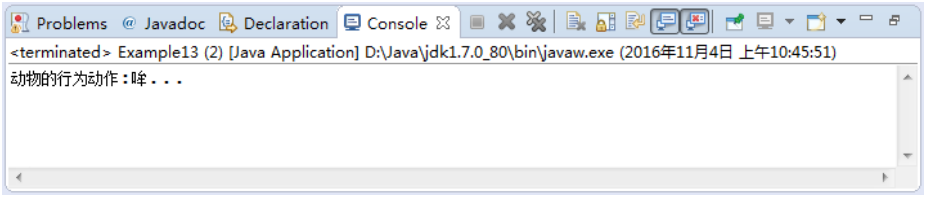

图3-16 运行结果

从图 3-16 的运行结果可以看出，Cow 类在实现了 Animal 接口后是可以被实例化的，并且实例化后就可以调用 Cow 类中的方法。需要注意的是，一个类实现一个接口，必须实现接口中所有的方法，如果不能实现某个方法，也必须写出一个空实现的方法。

为了加深读者对接口的了解，接下来对接口的特性进行讲解，具体如下：

- 接口的访问限定只有 public 和缺省的；
- interface 是声明接口的关键字，与 class 类似；
- 允许接口的多重继承，通过“extends 父接口名列表”可以继承多个接口；
- 对接口体中定义的常量，系统默认为是“public static final”修饰的，不需要指定；
- 对接口体中声明的方法，系统默认为是“public abstract”的；
- 接口中的方法都是抽象的，不能实例化对象。

3.9 多态

3.9.1 多态概述

所谓的多态就是在同一个方法中，由于参数类型不同而出现执行效果各异的现象。例如在实现动物叫

的方法中，由于每种动物叫声不同，因此可以在方法中接收一个动物类型的参数，当传入具体的某个动物时发出具体的叫声。在 Java 中为了实现多态，允许使用父类类型的变量来引用一个子类类型对象，根据子类对象特征不同，得到不同的运行结果。接下来通过一个案例来演示多态的使用，具体步骤如下：

1. 创建 Animal 接口

创建一个 Animal 接口，并在接口中定义一个抽象方法 call()，如文件 3-31 所示。

文件3-31 Animal.java

```
1  package com.itheima.example14;
2  //定义接口 Anmal
3  interface Animal {
4      void call();     // 定义抽象 call()方法
5  }
```

2. 创建 Cow 类

创建一个 Cow 类，使其实现 Animal 接口，并在 Animal 接口实现抽象方法 call()，如文件 3-32 所示。

文件3-32 Cow.java

```
1  package com.itheima.example14;
2  // 定义 Cow 类实现 Animal 接口
3  class Cow implements Animal {
4      // 实现 call()方法
5      public void call() {
6          System.out.println("哞...");
7      }
8  }
```

3. 创建 Sheep 类

创建一个 Sheep 类，与 Cow 类相似，同样实现了 Animal 接口，并实现了抽象方法 call()，如文件 3-33 所示。

文件3-33 Sheep.java

```
1  package com.itheima.example14;
2  // 定义 Sheep 类实现 Animal 接口
3  class Sheep implements Animal {
4      // 实现 shout()方法
5      public void call() {
6          System.out.println("咩...");
7      }
8  }
```

4. 创建测试类

创建一个测试类 Example14，在该类中通过父类类型变量引用不同的子类对象，当调用 animalCall() 方法时，将父类引用的两个不同子类对象分别传入该方法，如文件 3-34 所示。

文件3-34 Example14.java

```
1  package com.itheima.example14;
2  public class Example14 {
3      public static void main(String[] args) {
4          Animal a1 = new Cow();    // 创建 Cow 对象,使用 Animal 类型的变量 a1 引用
```

```
5           Animal a2 = new Sheep(); // 创建 Sheep 对象,使用 Animal 类型的变量 a2 引用
6           animalCall(a1); // 调用 animalCall()方法，将 a1 作为参数传入
7           animalCall(a2); // 调用 animalCall()方法，将 a2 作为参数传入
8       }
9       // 定义静态的 animalCall()方法，接收一个 Animal 类型的参数
10      public static void animalCall(Animal a) {
11          a.call(); // 调用实际参数的 call()方法
12      }
13  }
```

运行结果如图 3-17 所示。

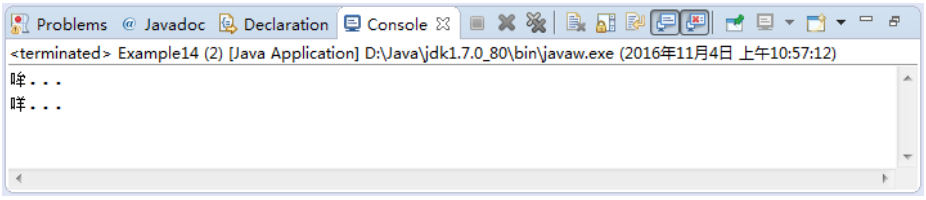

图3-17 运行结果

从图 3-17 的运行结果可以看出，两个不同的子类对象调用 call()时，分别打印出了“ 哞...”和“ 咩...”。

3.9.2 对象的类型转换

在多态的学习中，涉及到将子类对象当作父类类型使用的情况，此种情况在 Java 的语言环境中称之为“向上转型”，例如下面两行代码：

```
Animal a1 = new Cow(); // 将 Cow 对象当作 Animal 类型来使用
Animal a2 = new Sheep(); // 将 Sheep 对象当作 Animal 类型来使用
```

需要注意的是，将子类对象当作父类对象使用时不需要任何显式地转换，但此时不能通过父类变量去调用子类中的特有方法。

接下来通过一个案例来演示对象的类型转换错误的情况，具体步骤如下：

1. 创建 Animal 接口

创建一个 Animal 接口，并在接口中定义一个抽象方法 call()，如文件 3-35 所示。

文件3-35 Animal.java

```
1  package com.itheima.example15;
2  //定义 Animal 类接口
3  interface Animal {
4      void call(); // 定义抽象方法 call()
5  }
```

2. 创建 Cow 类

创建一个 Cow 类，使其实现 Animal 接口，并在 Animal 接口重写抽象方法 call()，如文件 3-36 所示。

文件3-36 Cow.java

```
1  package com.itheima.example15;
2  //定义 Cat 类实现 Animal 接口
3  public class Cow implements Animal {
4      // 实现 call()方法
```

```
5       public void call() {
6           System.out.println("哞...");
7       }
8   }
```

3. 创建 Sheep 类

创建一个 Sheep 类，与 Cow 类相似，同样实现了 Animal 接口，并重写了抽象方法 call()，如文件 3-37 所示。

文件3-37 Sheep.java

```
1   package com.itheima.example15;
2   //定义 Sheep 类实现 Animal 接口
3   public class Sheep implements Animal {
4       // 实现 call()方法
5       public void call() {
6           System.out.println("咩...");
7       }
8   }
```

4. 创建测试类

创建测试类 Example15 类，在该类中创建了一个 Sheep 对象，该对象作为参数的形式传入 animalCall()方法中。然后在 animalCall()方法中，将 Sheep 对象强制转换成 Cow 类型，并调用 call()方法，如文件 3-38 所示。

文件3-38 Example15.java

```
1   package com.itheima.example15;
2   // 定义测试类
3   public class Example15 {
4       public static void main(String[] args) {
5           Sheep s = new Sheep(); // 创建 Sheep 类型的实例对象
6           animalCall(s);          // 调用 animalCall()方法,将 Sheep 作为参数传入
7       }
8       // 定义静态方法 animalShout(),接收一个 Animal 类型的参数
9       public static void animalCall(Animal a) {
10          Cow c = (Cow) a; // 将 a 对象强制转换成 Cow 类型
11          c.call();        // 调用 Cow 的 call()方法
12      }
13  }
```

在文件 3-38 中，首先创建了一个 Sheep 对象，该对象作为参数的形式传入 animalCall()方法中。然后在 animalCall()方法中，将 Sheep 对象强制转换成 Cow 类型，并调用 call()方法，运行结果如图 3-18 所示。

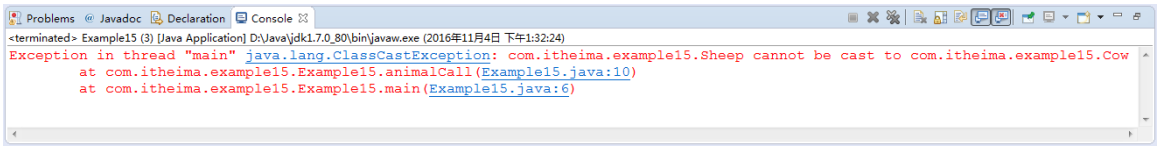

图3-18 运行结果

从图 3-18 的运行报错可以看出，提示 Sheep 类型不能转换成 Cow 类型。出错的原因是，在调用 animalCall()方法时，传入一个 Sheep 对象，在强制类型转换时，Animal 类型的变量无法强转为 Cow 类

型。

针对这种情况，Java 提供了一个关键字 instanceof，它可以判断一个对象是否为某个类(或接口)的实例或者子类实例，语法格式如下：

```
对象(或者对象引用变量) instanceof 类(或接口)
```

接下来对文件 3-38 的 animalCall ()方法进行修改，具体代码如下：

```
    public static void animalCall(Animal a) {
        if (a instanceof Cow) { // 判断 a 对象是否是 Cow 类的实例对象
            Cow c=(Cow)a;        // 将 a 对象强转为 Cow 类型
            c.call();           // 调用 Cow 的 call()方法
        } else {
            System.out.println("这个动物不是牛！");
        }
    }
```

运行结果如图 3-19 所示。

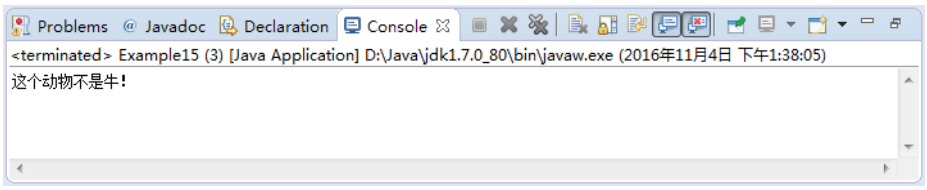

图3-19 运行结果

从上述代码中可以看出，使用 instanceof 关键字判断 animalCall()方法中传入的对象的类型，如果是 Cow 类型就进行强制类型转换，否则就打印"这个动物不是牛！"。该文件中，由于传入的对象为 Sheep 类型，因此出现图 3-19 的运行结果。

3.9.3 匿名内部类

在前面的多态讲解中，如果方法的参数被定义为一个接口类型，那么就需要定义一个类来实现接口，并根据该类进行对象实例化。除此之外，还可以使用匿名内部类来实现接口。当程序中使用匿名内部类时，在定义匿名内部类的地方往往直接创建该类的一个对象。

为了便于初学者理解，首先看一下匿名内部类的格式，具体如下：

```
new 父类(参数列表) 或 父接口(){
    //匿名内部类实现部分
}
```

为了让读者能更容易地理解匿名内部类的使用，接下来通过一个案例来学习匿名内部类的使用，具体步骤如下：

1. 创建 Animal 接口

创建一个 Animal 接口，并在接口中定义一个抽象方法 call()，如文件 3-39 所示。

文件3-39 Animal.java

```
1  package com.itheima.example16;
2  //定义动物类接口
3  interface Animal { // 定义动物类接口
4      void call();   // 定义方法 call()
5  }
```

2. 创建测试类

创建一个测试类 Example16，在该类中通过匿名内部类的形式来实现接口，如文件 3-40 所示。

文件3-40 Example16.java

```
1  package com.itheima.example16;
2  public class Example16 {
3      public static void main(String[] args) {
4          // 定义匿名内部类作为参数传递给 animalCall()方法
5          animalCall(new Animal() {
6              // 实现 call()方法
7              public void call() {
8                  System.out.println("哞...");
9              }
10         });
11     }
12     // 定义静态方法 animalCall()
13     public static void animalCall(Animal a) {
14         a.call(); // 调用传入对象 a 的 call()方法
15     }
16 }
```

在文件 3-40 中，调用 animalCall()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递给 animalCall()方法。在 new Animal()后面有一对大括号，表示创建了 Animal 的子类对象，该子类是匿名的，运行结果如图 3-20 所示。

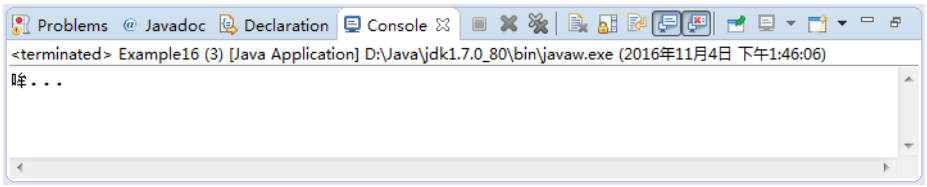

图3-20 运行结果

从图 3-20 的运行结果可以看出，使用了匿名内部类的方式实现 Animal 接口，调用 call()方法并打印输出结果为 “哞...”。

3.10 Exception（异常）

3.10.1什么是异常

在程序运行的过程中也会发生异常情况，例如运行时内存溢出、磁盘空间不足、网络中断等。针对此类状况，Java 中提供了异常处理机制，以异常类的形式对这些不正常情况进行封装，通过异常处理机制对程序代码发生的各种问题进行有针对性地处理。接下来通过一个案例来了解一下什么是异常，如文件 3-41 所示。

文件3-41 Example17.java

```
1  package com.itheima.example17;
2  public class Example17 {
```

```
3      public static void main(String[] args) {
4          int res = calculate(5, 0); // 调用 calculate()方法
5          System.out.println(res);
6      }
7      // 下面的方法实现了两个整数相除
8      public static int calculate(int a, int b) {
9          int res = a / b; // 定义一个变量 res 记录两个数相除的结果
10         return res;      // 将结果返回
11     }
12 }
```

运行结果如图 3-21 所示。

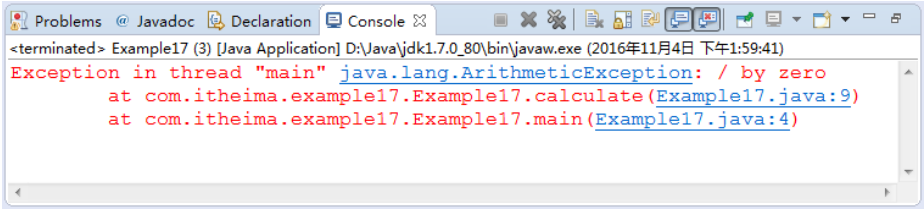

图3-21 运行结果

从图 3-21 的运行结果可以看出，程序出现了算术异常（ArithmeticException），这个异常是由于在第 4 行代码调用 divide()方法时传入了参数 0，而在 calculate()方法中，运算时出现了被 0 除的情况。程序出现异常后导致程序立即结束，无法继续向下执行。

3.10.2常见的异常类

在上一小节中产生的 ArithmeticException 异常只是 Java 异常体系中的一种，在 Java 中还提供了大量的异常类，这些异常类都是 java.lang.Throwable 类的子类。

接下来通过一张图来学习 Throwable 类的继承体系，如图 3-22 所示。

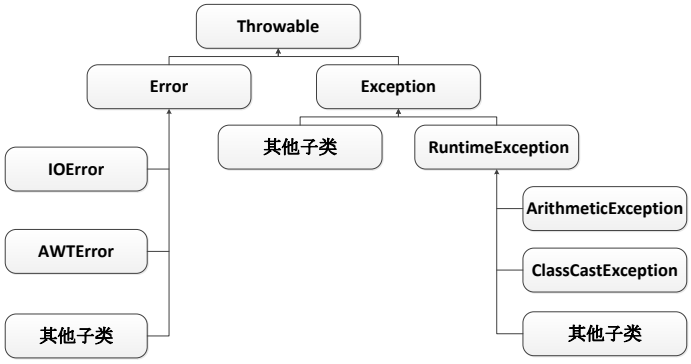

图3-22 Throwable 体系架构图

通过图 3-22 可以看出，Throwable 有两个直接子类 Error 和 Exception，其中 Error 表示程序代码中出现的错误，Exception 表示程序代码中出现的异常。他们的区别在于，错误是指仅靠程序本身是不能恢复执行的，而异常是指通过程序本身可以处理的错误。

3.10.3try…catch 和 finally

在文件 3-41 中，由于出现了异常导致程序代码无法继续向下执行。为了解决这样的问题，Java 中提供

了一种对异常进行处理的方式，即异常捕获。异常捕获通常使用的是 try…catch 语句，具体语法格式如下：

```
try{
    //程序代码块
}catch(异常类型(Exception 类或其子类) 变量名){
    //对异常的处理
}
```

上述代码中， try 代码块中用于编写可能发生异常的 Java 语句，catch 代码块中用于编写针对异常进行处理的代码。当程序发生异常时，系统会将异常信息封装成一个对象传递给 catch 代码块，catch 代码块需要一个 Exception 类型的参数来接收。

在处理异常时，我们偶尔也希望无论程序是否发生异常都要执行某个特定语句，此时就需要在 try…catch 后面加上一个 finally 语句，finally 语句中的内容无论程序是否发生异常都会执行。接下来使用 try...catch 和 finally 语句对文件 3-41 中出现的异常进行捕获，如文件 3-42 所示。

文件3-42 Example18.java

```
1  package com.itheima.example18;
2  public class Example18 {
3      public static void main(String[] args) {
4          // 下面的代码定义了一个 try…catch…finally 语句用于捕获异常
5          try {
6              int res = calculate(5, 0); // 调用 calculate()方法
7              System.out.println(res);
8          } catch (Exception e) {
9              System.out.println("捕获的异常是：" + e.getMessage());
10             return;
11         } finally {
12             System.out.println("finally 代码块");
13         }
14         System.out.println("程序代码继续执行...");
15     }
16     // 下面的方法实现了两个整数相除
17     public static int calculate(int a, int b) {
18         int res = a / b;    // 定义一个变量 res 记录两个数相除的结果
19         return res;         // 将结果返回
20     }
21 }
```

运行结果如图 3-23 所示。

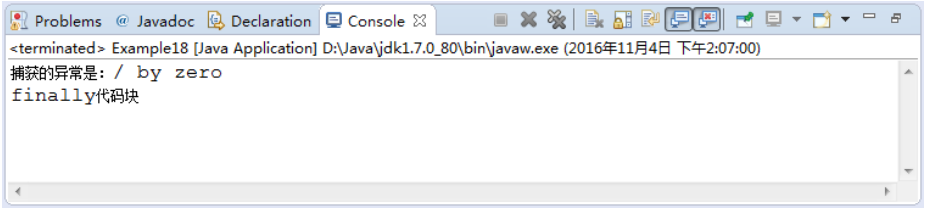

图3-23 运行结果

在文件 3-42 的 catch 代码块中增加了一个 return 语句，用于终止当前方法的执行，此时程序的第 14 行代码就不会执行了，但 finally 中的代码仍会执行，说明 finally 中的代码并不会被 return 语句所影响。由此可知，通常情况下会在 finally 代码块中来完成关闭系统资源等操作。

3.10.4throws 关键字

在上一节中已经了解到异常的发生可以使用 try…catch 语句的方式进行处理，除了此方法以外，在 Java 中还提供了另一种对异常的处理方式，即抛出异常。抛出异常使用 throws 关键字对外声明该方法有可能发生的异常，这样在调用方法时，就很清楚该方法是否有异常，并对异常进行针对性的处理，否则编译失败。具体语法格式如下：

```
修饰符 返回值类型 方法名([参数 1，参数 2.....])throws ExceptionType1[,ExceptionType2.....]{
}
```

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，通常将这种做法称为方法声明抛出一个异常。接下来对文件 3-42 进行修改，在 calculate()方法上声明抛出异常，如文件 3-43 所示。

文件3-43 Example19.java

```
1  package com.itheima.example19;
2  public class Example19 {
3      public static void main(String[] args) {
4          int res = calculate(6, 3); // 调用 calculate()方法
5          System.out.println(res);
6      }
7      // 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
8      public static int calculate(int a, int b) throws Exception {
9          int res = a / b;    // 定义一个变量 res 记录两个数相除的结果
10         return res;         // 将结果返回
11     }
12 }
```

在 Eclipse 中，编译程序报错，结果如图 3-24 所示。

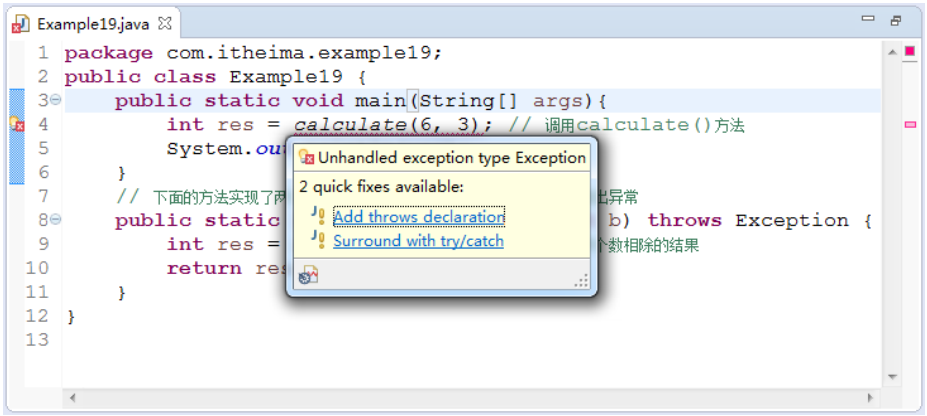

图3-24 编译结果

在图 3-24 中，第 4 行代码调用 calculate()方法时，虽然程序在运行时不会出现被 0 除的异常，但是由于定义 calculate()方法时声明抛出了异常，调用者在调用 calculate()方法时就必须进行处理，否则就会出现编译报错。下面对文件 3-43 进行修改，如文件 3-44 所示。

文件3-44 Example20.java

```
1  package com.itheima.example20;
2  public class Example20 {
3      public static void main(String[] args) throws Exception {
4          int res = calculate(6, 3); // 调用 calculate()方法
```

```
5           System.out.println(res);
6       }
7       // 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
8       public static int calculate(int x, int y) throws Exception {
9           int res = x / y;    // 定义一个变量 res 记录两个数相除的结果
10          return res;         // 将结果返回
11      }
12  }
```

运行结果如图 3-25 所示。

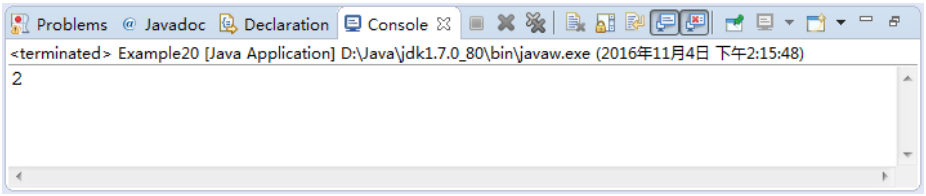

图3-25 运行结果

文件 3-44 中，使用 throws 关键字将异常抛出，这样程序就可以编译通过，运行后正确的打印出了运行结果 2。

注意：

在程序开发中有两种常见异常，一种是编译时异常，另一种是运行时异常，区别是编译时产生的异常必须进行处理否则编译不通过，而运行时异常是在程序运行中产生的，即使不做异常处理也能通过编译。

3.11 访问控制

在 Java 中，针对类、成员方法和属性提供了四种访问级别，分别是 private、default、protected 和 public。这四种访问级别控制了访问权限的大小，接下来通过一个图例对这四种访问级别进行排序，如图 3-26 所示。

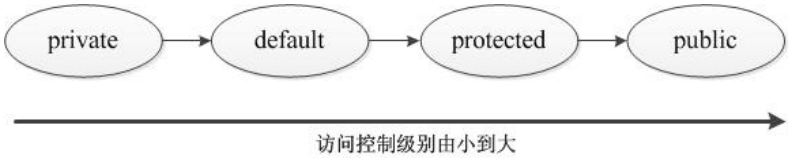

图3-26 访问级别

接下来针对图 3-26 中展示的四种访问控制级别进行介绍，具体如下：

- private(类访问级别) ：表示私有的，使用 private 修饰的成员只能被该类的其他成员访问，其他类无法直接访问。
- default(包访问级别) ：若没有访问修饰符，则系统默认使用 default。该类可以被本类包中的所有类访问。
- protected(子类访问级别) ：表示受保护的，该类可以被同一包下的其他类访问，也能被不同包下的子类所访问。
- public(公共访问级别) ：表示公有的，该类或者类中的成员可以被所有的类访问。

接下来通过一个表将这四种访问级别更加直观的表示出来，如表 3-1 所示。

表3-1 访问控制级别

访问范围	private	default	protected	public
同一类中	✓	✓	✓	✓

同一包中		✓	✓	✓
子类中			✓	✓
全局范围				✓