

第7章

面向对象进阶

本章读者可以学到如下实例：

- ▶ 实例 059 经理与员工的差异
- ▶ 实例 060 重写父类中的方法
- ▶ 实例 061 计算几何图形的面积
- ▶ 实例 062 简单的汽车销售商场
- ▶ 实例 063 使用 Comparable 接口自定义排序
- ▶ 实例 064 策略模式的简单应用
- ▶ 实例 065 适配器模式的简单应用
- ▶ 实例 066 普通内部类的简单应用
- ▶ 实例 067 局部内部类的简单应用
- ▶ 实例 068 匿名内部类的简单应用
- ▶ 实例 069 静态内部类的简单应用
- ▶ 实例 070 实例化 Class 类的几种方式
- ▶ 实例 071 查看类的声明
- ▶ 实例 072 查看类的成员
- ▶ 实例 073 查看内部类信息
- ▶ 实例 074 动态设置类的私有域
- ▶ 实例 075 动态调用类中方法
- ▶ 实例 076 动态实例化类
- ▶ 实例 077 创建长度可变的数组
- ▶ 实例 078 利用反射重写 `toString()` 方法



实例 059 经理与员工的差异

(实例位置: 配套资源\SL\07\059)



Note

实例说明

对于在同一家公司工作的经理和员工而言,两者是有很多共同点的。例如每个月都要发工资,但是经理在完成目标任务后,还会获得奖金。此时,利用员工类来编写经理类就会少写很多代码;利用继承技术可以让经理类使用员工类中定义的属性和方法。本实例将通过继承演示经理与员工的差异。实例的运行效果如图 7.1 所示。

实现过程

- (1) 在 Eclipse 中创建项目 059, 并在该项目中创建 com.mingrisoft 包。
- (2) 在 com.mingrisoft 包中创建类文件, 名称为 Employee。在该类中定义 3 个属性, 分别是 name(表示员工的姓名)、salary(表示员工的工资)和 birthday(表示员工的生日), 并分别为它们定义 getXXX() 和 setXXX() 方法。关键代码如下:

```
import java.util.Date;
public class Employee {
    private String name; //员工的姓名
    private double salary; //员工的工资
    private Date birthday; //员工的生日

    public String getName() { //获取员工的姓名
        return name;
    }
    public void setName(String name) { //设置员工的姓名
        this.name = name;
    }

    public double getSalary() { //获取员工的工资
        return salary;
    }
    public void setSalary(double salary) { //设置员工的工资
        this.salary = salary;
    }
    public Date getBirthday() { //获取员工的生日
        return birthday;
    }
    public void setBirthday(Date birthday) { //设置员工的生日
        this.birthday = birthday;
    }
}
```

- (3) 在 com.mingrisoft 包中再创建一个名称为 Manager 的类, 该类继承自 Employee。在



图 7.1 经理与员工的差异



该类中定义一个 bonus 域，表示经理的奖金，并为其设置 getXXX() 和 setXXX() 方法。关键代码如下：

```
public class Manager extends Employee {
    private double bonus; //经理的奖金
    public double getBonus() { //获得经理的奖金
        return bonus;
    }
    public void setBonus(double bonus) { //设置经理的奖金
        this.bonus = bonus;
    }
}
```

(4) 在 com.mingrisoft 包中再创建一个名称为 Text 的类，用于测试。在该类中分别创建 Employee 和 Manager 对象，并为其赋值，然后输出其属性。关键代码如下：

```
import java.util.Date; //导入 java.util.Date 类
public class Test {
    public static void main(String[] args) {
        Employee employee = new Employee(); //创建 Employee 对象并为其赋值
        employee.setName("Java");
        employee.setSalary(100);
        employee.setBirthday(new Date());
        Manager manager = new Manager(); //创建 Manager 对象并为其赋值
        manager.setName("明日科技");
        manager.setSalary(3000);
        manager.setBirthday(new Date());
        manager.setBonus(2000);
        //输出经理和员工的属性值
        System.out.println("员工的姓名：" + employee.getName());
        System.out.println("员工的工资：" + employee.getSalary());
        System.out.println("员工的生日：" + employee.getBirthday());
        System.out.println("经理的姓名：" + manager.getName());
        System.out.println("经理的工资：" + manager.getSalary());
        System.out.println("经理的生日：" + manager.getBirthday());
        System.out.println("经理的奖金：" + manager.getBonus());
    }
}
```



Note

指点迷津：

在 Manager 类中并未定义姓名等域，然而却可以使用，这就是继承的好处。

脚下留神：

虽然使用继承能少写很多代码，但是不要滥用继承。在使用继承前，需要考虑一下两者之间是否真的是“is-a”的关系，这是继承的重要特征。本实例中，经理显然是员工，所以可以用继承。另外，子类也可以成为其他类的父类，这样就构成了一棵继承树。

技术要点

在面向对象程序设计中，继承是其基本特性之一。在 Java 中，如果想表明类 A 继承了类 B，



可以使用下面的语法定义类 A:

```
public class A extends B {}
```

类 A 称为子类、派生类或孩子类，类 B 称为超类、基类或父类。尽管类 B 是一个超类，但是并不意味着类 B 比类 A 有更多的功能。相反，类 A 比类 B 拥有的功能更加丰富。



Note

指点迷津：

在继承树中，从下往上越来越抽象，从上往下越来越具体。

实例 060 重写父类中的方法

(实例位置：配套资源\SL\07\060)

实例说明

在继承了一个类之后，也就可以使用父类中定义的方法。然而，父类中的方法可能并不完全适用于子类。此时，如果不想定义新的方法，则可以重写父类中的方法。本实例将演示如何重写父类中的方法，实例的运行效果如图 7.2 所示。

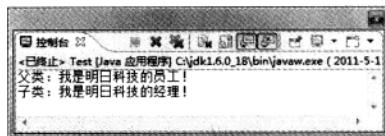


图 7.2 重写父类中的方法

实现过程

(1) 在 Eclipse 中创建项目 060，并在该项目中创建 com.mingrisoft 包。

(2) 在 com.mingrisoft 包中创建类文件，名称为 Employee。在该类中添加 getInfo()方法，返回值为字符串“父类：我是明日科技的员工！”。关键代码如下：

```
public class Employee {  
    public String getInfo() {  
        return "父类：我是明日科技的员工！";  
    }  
}
```

(3) 在 com.mingrisoft 包中再创建一个名称为 Manager 的类，该类继承自 Employee。在该类中重写 getInfo()方法。关键代码如下：

```
public class Manager extends Employee {  
    @Override  
    public String getInfo() {  
        return "子类：我是明日科技的经理！";  
    }  
}
```

指点迷津：

在 Java SE 5.0 版中新增了注解功能，@Override 是最常用的注解之一。该注解只能应用在方法上，可以测试该方法是否重写了父类中的方法，如果没有则会在编译时报错。使用该注解可以很好地避免重写时发生的各种问题，因此推荐读者使用。

(4) 在 com.mingrisoft 包中再创建一个名称为 Text 的类，用于测试。在该类中分别创建



Employee 和 Manager 对象，并分别输出 getInfo()方法的返回值。关键代码如下：

```
public class Test {
    public static void main(String[] args) {
        Employee employee = new Employee(); // 创建 Employee 对象
        System.out.println(employee.getInfo()); // 输出 Employee 对象的 getInfo()方法返回值
        Manager manager = new Manager(); // 创建 Manager 对象
        System.out.println(manager.getInfo()); // 输出 Manager 对象的 getInfo()方法返回值
    }
}
```



Note

技术要点

本实例主要应用的是方法的重写（Overriding）只能发生在存在继承关系的类中。重写方法需要注意以下几点：

- 重写方法与原来方法签名要相同，即方法名称和参数（包括顺序）要相同。
- 重写方法的可见性不能小于原来的方法。
- 重写方法抛出异常的范围不能大于原来方法抛出异常的范围。

实例 061 计算几何图形的面积

（实例位置：配套资源\SL\07\061）

实例说明

对于每个几何图形而言，都有一些共同的属性，如名字和面积等，而其计算面积的方法却各不相同。为了简化开发，本实例将定义一个超类来实现输出名字的方法，并使用抽象方法来计算面积。实例的运行效果如图 7.3 所示。

实现过程

- (1) 在 Eclipse 中创建项目 061，并在该项目中创建 com.mingrisoft 包。
- (2) 在 com.mingrisoft 包中创建一个抽象类，名称为 Shape。在该类中定义两个方法，一个是 getName()，用于使用反射机制获得类名称；另一个是抽象方法 getArea()，并未实现。关键代码如下：

```
public abstract class Shape {
    public String getName() { // 获得图形的名称
        return this.getClass().getSimpleName();
    }
    public abstract double getArea(); // 获得图形的面积
}
```

- (3) 在 com.mingrisoft 包中再创建一个名称为 Circle 的类，该类继承自 Shape，并实现了抽象方法 getArea()。在该类的构造方法中，获得了圆形的半径，用于在 getArea()中计算面积。关键代码如下：

```
public class Circle extends Shape {
    private double radius;
```

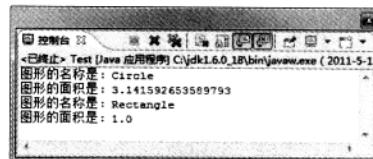


图 7.3 计算几何图形的面积



```
public Circle(double radius) { //获得圆形的半径
    this.radius = radius;
}
@Override
public double getArea() { //计算圆形的面积
    return Math.PI * Math.pow(radius, 2);
}
```



Note

(4) 在 com.mingrisoft 包中再创建一个名称为 Rectangle 的类，该类继承自 Shape，并实现了抽象方法 getArea()。在该类的构造方法中，获得了矩形的长和宽，用于在 getArea() 中计算面积。关键代码如下：

```
public class Rectangle extends Shape {
    private double length;
    private double width;
    public Rectangle(double length, double width) { //获得矩形的长和宽
        this.length = length;
        this.width = width;
    }
    @Override
    public double getArea() { //计算矩形的面积
        return length * width;
    }
}
```

(5) 在 com.mingrisoft 包中再创建一个名称为 Test 的类，用来进行测试，在该类中创建 Circle 和 Rectangle 对象，并分别输出图形的名称和面积。关键代码如下：

```
public class Test {
    public static void main(String[] args) {
        Circle circle = new Circle(1); //创建圆形对象并将半径设置成 1
        System.out.println("图形的名称是：" + circle.getName());
        System.out.println("图形的面积是：" + circle.getArea());
        Rectangle rectangle = new Rectangle(1, 1); //创建矩形对象并将长和宽设置成 1
        System.out.println("图形的名称是：" + rectangle.getName());
        System.out.println("图形的面积是：" + rectangle.getArea());
    }
}
```

指点迷津：

在抽象类中，可以定义抽象方法（使用 abstract 修饰的方法），也可以定义普通方法。包含抽象方法的类必须是抽象类，而抽象类不是必须包含抽象方法。对于抽象方法而言，仅定义一个声明即可，即抽象方法是没有方法体的。

技术要点

本实例应用的主要技术就是抽象类。下面对抽象类进行介绍。

在设计类的过程中，通常会将一些类所具有的公共属性和方法移到超类中，这样就不必重复定义了。然而这些类的超类却经常没有实际意义。通常将它设置成抽象的，这样可以避免创建该类的对象。声明一个最简单的抽象类的代码如下：

```
public abstract class Shape {}
```



参数说明

Shape: 为抽象类的类名。

脚下留神:

抽象类是不能直接实例化的，如果要获得该类的实例可以使用静态方法创建其实现类对象。



Note

实例 062 简单的汽车销售商场

(实例位置: 配套资源\SL\07\062)

实例说明

当顾客在商场购物时，卖家需要根据顾客的需求提取商品。对于汽车销售商场也是如此。用户需要先指定购买的车型，然后商家去提取该车型的汽车。本实例将实现一个简单的汽车销售商场，用来演示多态的用法。实例的运行效果如图 7.4 所示。

实现过程

(1) 在 Eclipse 中创建项目 062，并在该项目中创建 com.mingrisoft 包。

(2) 在 com.mingrisoft 包中新建一个抽象类，名称为 Car，在该类中定义一个抽象方法 getInfo()。关键代码如下：

```
public abstract class Car {
    public abstract String getInfo();           //用来描述汽车的信息
}
```

(3) 在 com.mingrisoft 包中再创建一个名称为 BMW 的类，该类继承自 Car 并实现其 getInfo() 方法。关键代码如下：

```
public class BMW extends Car {
    @Override
    public String getInfo() {                  //用来描述汽车的信息
        return "BMW";
    }
}
```

(4) 在 com.mingrisoft 包中再创建一个名称为 Benz 的类，该类继承自 Car 并实现其 getInfo() 方法。关键代码如下：

```
public class Benz extends Car {
    @Override
    public String getInfo() {                  //用来描述汽车的信息
        return "Benz";
    }
}
```

(5) 在 com.mingrisoft 包中再创建一个名称为 CarFactory 的类，该类定义了一个静态方法 getCar()，它可以根据用户指定的车型来创建对象。关键代码如下：

```
public class CarFactory {
    public static Car getCar(String name) {
```

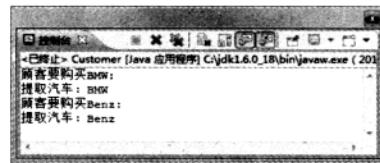


图 7.4 简单的汽车销售商场



(6) 在 com.mingrisoft 包中再创建一个名称为 Customer 的类, 用来进行测试。在 main() 方法中, 根据用户的需要提取了不同的汽车。关键代码如下:

```
public class Customer {  
    public static void main(String[] args) {  
        System.out.println("顾客要购买 BMW:");  
        Car bmw = CarFactory.getCar("BMW");  
        System.out.println("提取汽车: " + bmw.getInfo());  
        System.out.println("顾客要购买 Benz:");  
        Car benz = CarFactory.getCar("Benz");  
        System.out.println("提取汽车: " + benz.getInfo());  
    }  
}
```

技术要点

本实例应用的主要技术就是面向对象程序设计中的多态。多态是面向对象程序设计的基本特性之一。使用多态的好处就是可以屏蔽对象之间的差异，从而增强了软件的扩展性和重用性。Java 的多态主要是通过重写父类（或接口）中的方法来实现的。对于香蕉、桔子等水果而言，人们通常关心其能吃的特性。如果说香蕉能吃、桔子能吃，则当再增加新的水果种类，如菠萝时还要写个菠萝能吃，这是非常麻烦的。使用多态的话可以写成水果能吃，当需要用到具体的水果时，系统会自动帮忙替换，从而简化开发。

实例 063 使用 Comparable 接口自定义排序

(实例位置: 配套资源\SL\07\063)

实例说明

默认情况下,保存在 List 集合中的数组是不进行排序的,不过可以通过使用 Comparable 接口自定义排序规则并自动排序。本实例将介绍如何使用 Comparable 接口自定义排序规则并自动排序。实例的运行效果如图 7.5 所示。

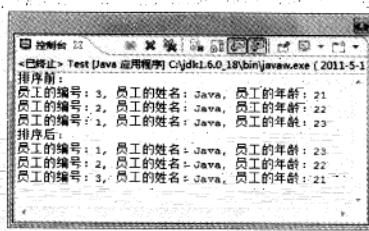


图 7.5 使用 Comparable 接口自定义排序

实现过程

(F) 在 Eclipse 中创建项目 063，并在该项目中创建 com.mingrisoft 包。



(2) 在 com.mingrisoft 包中新建一个 Java 类, 名称为 Employee。在该类中首先定义 3 个属性, 分别是 id(表示员工的编号)、name(表示员工的姓名)和 age(表示员工的年龄), 然后在构造方法中初始化这 3 个属性, 最后再实现接口中定义的 compareTo()方法, 将对象按编号升序排列。关键代码如下:

```

public class Employee implements Comparable<Employee> {
    private int id;                                //员工的编号
    private String name;                            //员工的姓名
    private int age;                               //员工的年龄
    public Employee(int id, String name, int age) { //利用构造方法初始化各个域
        this.id = id;
        this.name = name;
        this.age = age;
    }
    @Override
    public int compareTo(Employee o) {             //利用编号实现对象间的比较
        if (id > o.id) {
            return 1;
        } else if (id < o.id) {
            return -1;
        }
        return 0;
    }
    @Override
    public String toString() {                     //重写 toString()方法
        StringBuilder sb = new StringBuilder();
        sb.append("员工的编号: " + id + ", ");
        sb.append("员工的姓名: " + name + ", ");
        sb.append("员工的年龄: " + age);
        return sb.toString();
    }
}

```

**Note**

指点迷津:

重写接口中的方法时, 要将访问权限限定符设为 public, 因为接口中的方法默认就是 public 的。

(3) 在 com.mingrisoft 包中再新建一个名称为 Test 的类, 用于进行测试。在该类中首先定义一个 List 集合来保存 3 个 Employee 对象, 并通过遍历输出集合中的元素, 再通过 Collections.sort() 方法执行自动排序, 最后再通过遍历输出排序后的集合中的元素。关键代码如下:

```

public class Test {
    public static void main(String[] args) {
        List<Employee> list=new ArrayList<Employee>();
        list.add(new Employee(3,"Java",21));
        list.add(new Employee(2,"Java",22));
        list.add(new Employee(1,"Java",23));
        System.out.println("排序前: ");
        for (Employee employee : list) {
            System.out.println(employee);
        }
    }
}

```



```

        System.out.println("排序后: ");
        Collections.sort(list); //执行自动排序
        for (Employee employee : list) {
            System.out.println(employee);
        }
    }
}

```



Note

技术要点

本实例主要应用 java.lang 包中的 Comparable 接口来实现自定义排序规则。Comparable 接口用于强行对实现它的每个类的对象进行整体排序。在实现该接口的类中，必须实现该接口中定义的 compareTo()方法，用于指定排序规则。Comparable 接口的定义如下：

```

public interface Comparable<T> {
    int compareTo(T other);
}

```

如果一个类要实现这个接口，可以使用如下语句声明：

```
public class Employee implements Comparable<Employee> {}
```

在 Employee 中必须实现接口中定义的 compareTo()方法。实现该方法后，如果将该对象保存到列表中，那么可以通过执行 Collections.sort()方法进行自动排序；如果保存到数组中，那么可以通过执行 Arrays.sort()方法进行自动排序。

指点迷津：

如果不想要在接口中定义的方法，则可以将类声明为抽象类，将接口中定义的方法声明为抽象方法。

实例 064 策略模式的简单应用

(实例位置：配套资源\SL\07\064)

实例说明

在使用图像处理软件处理图片后，需要选择一种格式进行保存，然而各种格式在底层实现的算法并不相同，这刚好适合策略模式。本实例将演示如何使用策略模式与简单工厂模式组合进行实例开发。实例的运行效果如图 7.6 所示。

实现过程

- (1) 在 Eclipse 中创建项目 064，并在该项目中创建 com.mingrisoft 包。
- (2) 在 com.mingrisoft 包中编写接口 ImageSaver，在该接口中定义 save()方法。关键代码如下：

```

public interface ImageSaver {
    void save(); //定义 save()方法
}

```

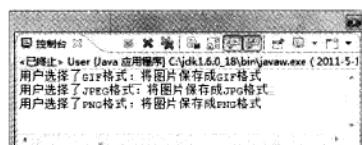


图 7.6 策略模式的简单应用



(3) 在 com.mingrisoft 包中再编写类 GIFSaver，该类实现了 ImageSaver 接口。在实现 save() 方法时将图片保存为 GIF 格式，关键代码如下：

```
public class GIFSaver implements ImageSaver {
    @Override
    public void save() { //实现 save()方法
        System.out.println("将图片保存成 GIF 格式");
    }
}
```



Note

指点迷津：

对于将图片保存成其他格式与存储为 GIF 格式类似，这里就不再赘述。

(4) 在 com.mingrisoft 包中再编写类 TypeChooser，该类根据用户提供的图片类型来选择合适的图片存储方式。关键代码如下：

```
public class TypeChooser {
    public static ImageSaver getSaver(String type) {
        if (type.equalsIgnoreCase("GIF")) { //使用 if...else 语句来判断图片的类型
            return new GIFSaver();
        } else if (type.equalsIgnoreCase("JPEG")) {
            return new JPEGSaver();
        } else if (type.equalsIgnoreCase("PNG")) {
            return new PNGSaver();
        } else {
            return null;
        }
    }
}
```

指点迷津：

此处使用了简单工厂模式，根据描述图片类型的字符串创建相应的图片保存类的对象。

(5) 在 com.mingrisoft 包中再编写类 User，该类模拟用户的操作，为类型选择器提供图片的类型。关键代码如下：

```
public class User {
    public static void main(String[] args) {
        System.out.print("用户选择了 GIF 格式： ");
        ImageSaver saver = TypeChooser.getSaver("GIF"); //获得保存图片为 GIF 类型的对象
        saver.save();
        System.out.print("用户选择了 JPEG 格式： "); //获得保存图片为 JPEG 类型的对象
        saver = TypeChooser.getSaver("JPEG");
        saver.save();
        System.out.print("用户选择了 PNG 格式： "); //获得保存图片为 PNG 类型的对象
        saver = TypeChooser.getSaver("PNG");
        saver.save();
    }
}
```



技术要点

本实例应用的最重要的技术就是策略模式。对于策略模式而言，需要定义一个接口或者抽象类来表示各种策略的抽象，这样就可以使用多态来让虚拟机选择不同的实现类。然后让每一种具体的策略来实现这个接口或继承抽象类，并为其中定义的方法提供具体的实现。由于在选择适当的策略上有些不方便，需要不断地判断需要的类型，因此用简单工厂方法来实现判断过程。



Note

实例 065 适配器模式的简单应用

(实例位置：配套资源\SL\07\065)

实例说明

对于刚从工厂生产出来的商品，有些功能并不能完全满足用户的需要。因此，用户通常会对其进行一定的改装工作。本实例将为普通的汽车增加 GPS 定位功能，借此演示适配器模式的用法。实例的运行效果如图 7.7 所示。

实现过程

- (1) 在 Eclipse 中创建项目 065，并在该项目中创建 com.mingrisoft 包。
- (2) 在 com.mingrisoft 包中编写类 Car，在该类中，首先定义两个属性，一个是 name，表示汽车的名字；另一个是 speed，表示汽车的速度。并为其提供 getXXX()和 setXXX()方法，然后通过重写 toString()方法来方便输出 Car 对象。关键代码如下：

```
public class Car {  
    private String name; //表示名称  
    private double speed; //表示速度  
    //省略 getXXX()和 setXXX()方法  
    @Override  
    public String toString() { //重写 toString()方法  
        StringBuilder sb = new StringBuilder();  
        sb.append("车名：" + name + ", ");  
        sb.append("速度：" + speed + "千米/小时");  
        return sb.toString();  
    }  
}
```

- (3) 在 com.mingrisoft 包中再编写接口 GPS，该接口中定义了 getLocation()方法，用来确定汽车的位置。关键代码如下：

```
public interface GPS {  
    Point getLocation(); //提供定位功能  
}
```

- (4) 在 com.mingrisoft 包中再编写类 GPSCar，该类继承 Car 并实现 GPS 接口。在该类中首先实现 getLocation()方法，用于实现确定汽车位置的功能，然后重写 toString()方法方便输出

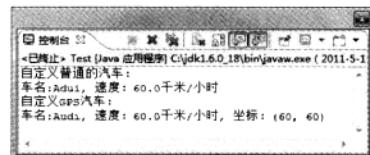


图 7.7 适配器模式的简单应用



GPSCar 对象。关键代码如下：

```
public class GPSCar extends Car implements GPS {
    @Override
    public Point getLocation() { //利用汽车的速度来确定汽车的位置
        Point point = new Point();
        point.setLocation(super.getSpeed(), super.getSpeed());
        return point;
    }
    @Override
    public String toString() { //重写 toString()方法
        StringBuilder sb = new StringBuilder();
        sb.append(super.toString());
        sb.append(", 坐标: (" + getLocation().x + ", " + getLocation().y + ")");
        return sb.toString();
    }
}
```

指点迷津：

可以使用 `super` 关键字调用父类中定义的方法。



Note

(5) 在 com.mingrisoft 包中再编写类 Test 进行测试。在该类中，分别创建 Car 和 GPSCar 对象，并对其初始化，然后输出这两个对象。关键代码如下：

```
public class Test {
    public static void main(String[] args) {
        System.out.println("自定义普通的汽车：");
        Car car = new Car(); //创建普通的汽车对象并初始化
        car.setName("Audi");
        car.setSpeed(60);
        System.out.println(car);
        System.out.println("自定义 GPS 汽车：");
        GPSCar gpsCar = new GPSCar(); //创建带 GPS 功能的汽车对象并初始化
        gpsCar.setName("Audi");
        gpsCar.setSpeed(60);
        System.out.println(gpsCar);
    }
}
```

技术要点

适配器模式可以在符合 OCP 原则（开闭原则）的基础上，为类增加新的功能。该模式涉及的角色主要有以下 3 个。

- 目标角色：就是期待得到的接口，如本实例的 GPS 接口。
 - 源角色：需要被增加功能的类或接口，如本实例的 Car 类。
 - 适配器角色：新创建的类，在源角色的基础上实现了目标角色，如本实例的 GPSCar 类。
- 关于各个类的继承（实现）关系如图 7.8 所示。

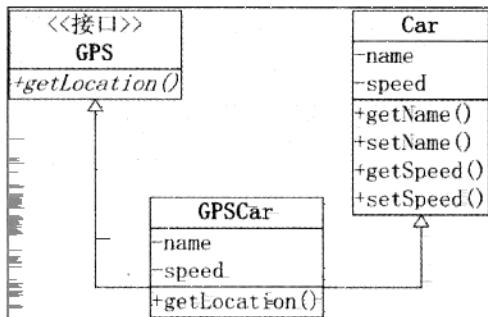
**Note**

图 7.8 适配器模式 UML 图

实例 066 普通内部类的简单应用

(实例位置：配套资源\SL\07\066)

实例说明

在使用图形界面程序时，用户总是希望界面是丰富多彩的，这就要求程序员根据不同的情况为界面设置不同的颜色。本实例定义了 3 个按钮，用户通过单击不同的按钮，可以为面板设置不同的颜色。运行本实例，将显示如图 7.9 所示的效果，单击“红色”按钮，即可将背景设置为红色；单击“绿色”按钮，即可将背景设置为绿色；单击“蓝色”按钮，即可将背景设置为蓝色，如图 7.10 所示。

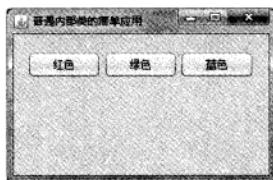


图 7.9 默认的运行效果

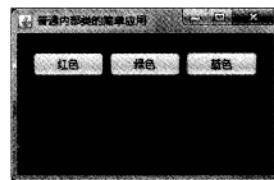


图 7.10 单击“蓝色”按钮的效果

实现过程

- (1) 在 Eclipse 中创建项目 066，并在该项目中创建 com.mingrisoft 包。
- (2) 在 com.mingrisoft 包中编写类 ButtonTest，该类继承自 JFrame。在该窗体中添加 3 个按钮，分别用来为面板设置不同的颜色。
- (3) 在 com.mingrisoft 包中再编写类 ColorAction，该类继承自 ActionListener 接口。在该类的构造方法中，需要为其指定一种颜色，在 actionPerformed()方法中将面板设置成指定的颜色。关键代码如下：

```

private class ColorAction implements ActionListener {
    private Color background;
    public ColorAction(Color background) {
        this.background = background;
    }
    @Override
  
```



```

public void actionPerformed(ActionEvent e) {
    contentPane.setBackground(background);
}

}
}

```

指点迷津：

panel 是在外部类 ButtonTest 中定义的域，但是在内部类中却可以直接使用。

**Note****技术要点**

在类中，除了可以定义域、方法和块，还可以定义类，这种类称为内部类。声明一个最简单的内部类的语法如下：

```

public class Outer {
    class Inner {}
}

```

内部类可以使用外部类中定义的属性和方法，即使它们都是私有的。编译器在编译内部类时，将内部类命名为 Outer\$Inner 的形式，虚拟机并不知道有内部类。

实例 067 局部内部类的简单应用

(实例位置：配套资源\SL\07\067)

实例说明

日常生活中，闹钟的应用非常广泛。使用它可以更好地帮助人们安排时间。本实例将实现一个非常简单的闹钟。运行本实例，控制台会不断输出当前的时间，并且每隔一秒钟会发出提示音。用户可以单击“确定”按钮来退出程序。实例的运行效果如图 7.11 所示。

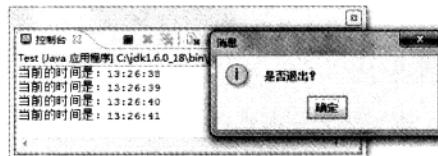


图 7.11 局部内部类的简单应用

实现过程

- (1) 在 Eclipse 中创建项目 067，并在该项目中创建 com.mingrisoft 包。
- (2) 在 com.mingrisoft 包中编写 Java 类，名称为 AlarmClock。在该类中，首先定义两个属性，一个是 delay，表示延迟的时间；另一个是 flag，表示是否需要发出提示声音。然后在 start()方法中，使用 Timer 类来安排动作发出事件。关键代码如下：

```

public class AlarmClock {
    private int delay; // 表示延迟时间
    private boolean flag; // 表示是否要发出声音
    public AlarmClock(int delay, boolean flag) { // 使用构造方法初始化各个域
        this.delay = delay;
        this.flag = flag;
    }
    public void start() {
}

```

**Note**

```

class Printer implements ActionListener {           // 定义内部类实现动作监听接口
    @Override
    public void actionPerformed(ActionEvent e) {
        SimpleDateFormat format = new SimpleDateFormat("k:m:s"); // 定义时间的格式
        String result = format.format(new Date());                // 获得当前的时间
        System.out.println("当前的时间是：" + result);           // 显示当前的时间
        if (flag) {                                              // 根据 flag 来决定是否要发出声音
            Toolkit.getDefaultToolkit().beep();
        }
    }
}
new Timer(delay, new Printer()).start();          // 创建 Timer 对象并启动
}
}

```

脚下留神：

如果 Printer 类使用了在 start()方法内定义的其他变量，则该变量也必须是 final 的。

(3) 编写类 Test 进行测试，在该类的 main()方法中创建 AlarmClock 对象，并调用其 start()方法。使用对话框提示用户是否要退出程序。关键代码如下：

```

public class Test {
    public static void main(String[] args) {
        AlarmClock clock = new AlarmClock(1000, true);           // 创建 AlarmClock 对象
        clock.start();                                            // 启动 start() 方法
        JOptionPane.showMessageDialog(null, "是否退出？");
        System.exit(0);                                           // 退出程序
    }
}

```

技术要点

本实例的技术要点就是局部内部类。在 Java 中可以将类定义在方法的内部，称为局部内部类。这种类不能使用 public 和 private 修饰，它的作用域被限定在声明这个类的方法中。局部内部类比其他内部类还有一个优点，就是可以访问方法参数。一个最简单的局部内部类代码如下：

```

public void book () {
    public class MingriSoft {}
}

```

脚下留神：

被局部内部类使用的方法参数必须是 final 的。

实例 068 匿名内部类的简单应用

(实例位置：配套资源\SL\07\068)

实例说明

在查看数码相片时，通常会使用一款图片查看软件，该软件应该能够遍历文件夹下的所有图



片并进行显示。本实例将编写一个非常简单的图片查看软件，它可以支持 6 张图片。通过单击不同的按钮就可以查看不同的图片。运行本实例，单击不同的按钮将显示不同的图片，例如，单击“图片 4”按钮，将显示如图 7.12 所示的图片；单击“图片 5”按钮，将显示如图 7.13 所示的图片。

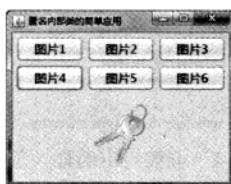


图 7.12 单击“图片 4”按钮的运行结果



图 7.13 单击“图片 5”按钮的运行结果

**Note**

实现过程

- (1) 在 Eclipse 中创建项目 068，并在该项目中创建 com.mingrisoft 包。
- (2) 在 com.mingrisoft 包中编写类，名称为 ImageViewer，该类继承自 JFrame。在窗体中添加 6 个按钮和一个标签，单击不同的按钮可以在标签上显示不同的图片。关键代码如下：

```
public ImageViewer() {
    //省略与 button1 无关的代码
    JButton button1 = new JButton("图片 1"); //创建按钮
    button1.setFont(new Font("微软雅黑", Font.PLAIN, 16)); //修改按钮上文本的字体
    button1.addActionListener(new ActionListener() { //为按钮增加监听器
        @Override
        public void actionPerformed(ActionEvent e) {
            label.setIcon(new ImageIcon("src/images/1.png")); //在标签中显示图片
        }
    });
    //省略与 button1 无关的代码
}
```

技术要点

本实例的技术要点就是匿名内部类。当只需要创建类的一个对象时，可以使用匿名内部类。 ActionListener 是 Swing 中动作事件的监听器，如果创建该接口的匿名内部类，可以使用以下代码：

```
ActionListener listener = new ActionListener() {
    public void actionPerformed(ActionEvent e) {}
};
```

脚下留神：

不要忘记写最后的“;”分号，这是语句的结束标识，和内部类无关。

实例 069 静态内部类的简单应用

(实例位置：配套资源\SL\07\069)

实例说明

当对元素进行排序时，需要明确各个元素如何比较大小。使用既定的比较方式，就可以求



出一个数组中的最大值和最小值。本实例使用静态内部类来实现使用一次遍历求最大值和最小值。实例的运行效果如图 7.14 所示。



Note

实现过程

(1) 在 Eclipse 中创建项目 069，并在该项目中创建 com.mingrisoft 包。

(2) 在 com.mingrisoft 包中编写 Java 类，名称为 MaxMin，在该类中，首先定义一个静态

内部类 Result，然后在该类中定义两个浮点型属性，一个是 max，表示最大值；另一个是 min，表示最小值。再使用构造方法为其初始化，并提供 getXXX()方法来获得这两个值。最后定义一个静态方法 getResult()，该方法的返回值是 Result 类型，这样就可以既保存最大值，又保存最小值。关键代码如下：

```
public class MaxMin {  
    public static class Result {  
        private double max; //表示最大值  
        private double min; //表示最小值  
        public Result(double max, double min) { //使用构造方法进行初始化  
            this.max = max;  
            this.min = min;  
        }  
        public double getMax() { //获得最大值  
            return max;  
        }  
        public double getMin() { //获得最小值  
            return min;  
        }  
    }  
    public static Result getResult(double[] array) {  
        double max = Double.MIN_VALUE; //遍历数组获得最大值和最小值  
        double min = Double.MAX_VALUE;  
        for (double i : array) {  
            if (i > max) {  
                max = i;  
            }  
            if (i < min) {  
                min = i;  
            }  
        }  
        return new Result(max, min); //返回 Result 对象  
    }  
}
```

(3) 在 com.mingrisoft 包中再编写类 Test 进行测试，在该类的 main()方法中，使用随机数初始化了一个长度为 5 的数组，并求得该数组的最大值和最小值。关键代码如下：

```
public class Test {  
    public static void main(String[] args) {  
        double[] array = new double[5];
```

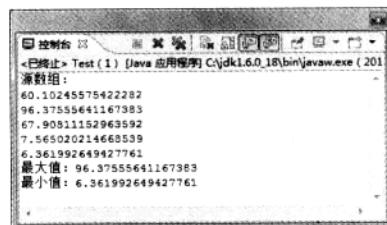


图 7.14 求数组中的最大值和最小值



```

for (int i = 0; i < array.length; i++) { // 初始化数组
    array[i] = 100 * Math.random();
}
System.out.println("源数组: ");
for (int i = 0; i < array.length; i++) { // 显示数组中的各个元素
    System.out.println(array[i]);
}
System.out.println("最大值: " + MaxMin.getResult(array).getMax()); // 显示最大值
System.out.println("最小值: " + MaxMin.getResult(array).getMin()); // 显示最小值
}
}

```

技术要点

本实例的技术要点就是静态内部类。静态内部类是使用 static 修饰的内部类，在静态内部类中，可以使用外部类定义的静态域，但是不能使用非静态域。这是静态内部类与非静态内部类的重要区别。定义一个最简单的静态内部类的代码如下：

```

public void book () {
    public static class MingriSoft {}
}

```

脚下留神：

不要将 MingriSoft 类声明成 private 的，否则不能使用其中定义的方法。

实例 070 实例化 Class 类的几种方式

(实例位置：配套资源\SL\07\070)

实例说明

Java 的数据类型可以分成两类，即引用类型和原始类型。无论哪种类型的对象，Java 虚拟机都会实例化不可变的 java.lang.Class 对象。它提供了在运行时检查对象属性的方法，这些属性包括它的成员和类型信息。更重要的是 Class 对象是所有反射 API 的入口。本实例将演示如何获得 Class 对象。实例的运行效果如图 7.15 所示。

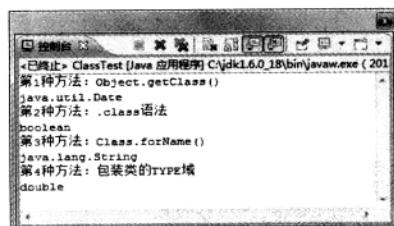


图 7.15 实例化 Class 类的几种方式

多学两招：

Class 类是泛型类，可以使用 @SuppressWarnings("unchecked") 忽略泛型或者使用 Class<?> 类型。

实现过程

(1) 在 Eclipse 中创建项目 070，并在该项目中创建 com.mingrisoft 包。





(2) 在 com.mingrisoft 包中编写类 ClassTest，在该类的 main()方法中，演示各种获得 Class 对象的方法。关键代码如下：

```
public class ClassTest {  
    @SuppressWarnings("unchecked")  
    public static void main(String[] args) throws ClassNotFoundException {  
        System.out.println("第 1 种方法： Object.getClass()");  
        Class c1 = new Date().getClass(); //使用 getClass()方法获得 Class 对象  
        System.out.println(c1.getName()); //输出对象名称  
        System.out.println("第 2 种方法： .class 语法");  
        Class c2 = boolean.class; //使用 .class 语法获得 Class 对象  
        System.out.println(c2.getName()); //输出对象名称  
        System.out.println("第 3 种方法： Class.forName()");  
        Class c3 = Class.forName("java.lang.String"); //使用 Class.forName()方法获得 Class 对象  
        System.out.println(c3.getName()); //输出对象名称  
        System.out.println("第 4 种方法： 包装类的 TYPE 域");  
        Class c4 = Double.TYPE; //使用 包装类获得 Class 对象  
        System.out.println(c4.getName()); //输出对象名称  
    }  
}
```

技术要点

本实例的技术要点就是如何获得 Class 对象。通常有以下 4 种方式可以获得 Class 对象。

- Object.getClass(): 如果一个类的对象可用，则最简单的获得 Class 的方法是使用 Object.getClass()。当然，这种方式只对引用类型有用。
- .class 语法：如果类型可用，但没有对象则可以在类型后加上“.class”来获得 Class 对象。这也是使原始类型获得 Class 对象的最简单的方式。
- Class.forName(): 如果知道类的全名，则可以使用静态方法 Class.forName()来获得 Class 对象。该方法不能用在原始类型上，但是可以用在原始类型数组上。

脚下留神：

Class.forName()方法会抛出 ClassNotFoundException 异常。

- 包装类的 TYPE 域：每个原始类型和 void 都有包装类，利用其 TYPE 域就可以获得 Class 对象。

实例 071 查看类的声明

(实例位置：配套资源\SL\07\071)

实例说明

通常类的声明包括常见修饰符（public、protected、private、abstract、static、final 和 strictfp 等）、类的名称、类的泛型参数、类的继承类（实现的接口）和类的注解等。本实例将演示如何用反射获得这些信息。实例的运行效果如图 7.16 所示。



Note



图 7.16 查看类的声明

实现过程

(1) 在 Eclipse 中创建项目 071，并在该项目中创建 com.mingrisoft 包。

(2) 在 com.mingrisoft 包中编写类 ClassDeclarationViewer，在 main()方法中输出了与类声明相关的各个项。关键代码如下：

```
public class ClassDeclarationViewer {
    public static void main(String[] args) throws ClassNotFoundException {
        Class<?> clazz = Class.forName("java.util.ArrayList"); //获得 ArrayList 类对象
        System.out.println("类的标准名称: " + clazz.getCanonicalName());
        System.out.println("类的修饰符: " + Modifier.toString(clazz.getModifiers()));
        //输出类的泛型参数
        TypeVariable<?>[] typeVariables = clazz.getTypeParameters();
        System.out.print("类的泛型参数: ");
        if (typeVariables.length != 0) {
            for (TypeVariable<?> typeVariable : typeVariables) {
                System.out.println(typeVariable + "\t");
            }
        } else {
            System.out.println("空");
        }
        //输出类所实现的所有接口
        Type[] interfaces = clazz.getGenericInterfaces();
        System.out.print("类所实现的接口: ");
        if (interfaces.length != 0) {
            for (Type type : interfaces) {
                System.out.println("\t" + type);
            }
        } else {
            System.out.println("\t" + "空");
        }
        //输出类的直接继承类，如果是继承自 Object 则返回空
        Type superClass = clazz.getGenericSuperclass();
        System.out.print("类的直接继承类: ");
        if (superClass != null) {
            System.out.println(superClass);
        } else {
            System.out.println("空");
        }
    }
}
```



Note

```
//输出类的所有注释信息，有些注释信息是不能用反射获得的
Annotation[] annotations = clazz.getAnnotations();
System.out.print("类的注解: ");
if (annotations.length != 0) {
    for (Annotation annotation : annotations) {
        System.out.println("\t" + annotation);
    }
} else {
    System.out.println("空");
}
}
```

多学两招：

通常只能通过 API 来查看类的定义，不过 Java 反射还提供了另一种方式来获得类的信息，读者也可以在程序中使用这些信息。另外，使用 `getInterfaces()` 方法也可以获得对象类的所有接口，但是不包含泛型信息。即使是 `getSuperclass()` 方法也不能获得有泛型信息的父类。

技术要点

`Class` 类的实例表示正在运行的 Java 应用程序中的类和接口。枚举是一种类，注释是一种接口。每个数组属于被映射为 `Class` 对象的一个类，所有具有相同元素类型和维数的数组都共享该 `Class` 对象。基本的 Java 类型（`boolean`、`byte`、`char`、`short`、`int`、`long`、`float` 和 `double`）和关键字 `void` 也表示为 `Class` 对象。它没有公共构造方法。`Class` 对象是在加载类时由 Java 虚拟机以及通过调用类加载器中的 `defineClass()` 方法自动构造的。本实例使用的方法如表 7.1 所示。

表 7.1 `Class` 类的常用方法

方 法 名	作 用
<code>forName(String className)</code>	根据给定的名称获得 <code>Class</code> 对象
<code>getAnnotations()</code>	返回此 <code>Class</code> 对象上存在的注释
<code>getCanonicalName()</code>	返回 Java Language Specification 中所定义的底层类的规范化名称
<code>getGenericInterfaces()</code>	返回泛型形式的对象类所实现的接口
<code>getGenericSuperclass()</code>	返回泛型形式的对象类所直接继承的超类
<code>getModifiers()</code>	返回此类或接口以整数编码的 Java 语言修饰符
<code>getTypeParameters()</code>	按声明顺序返回 <code>TypeVariable</code> 对象的一个数组

实例 072 查看类的成员

（实例位置：配套资源\SL\07\072）

实例说明

在一个类的内部，一般包括成员变量、构造方法、普通方法和内部类等。使用反射机制可以在无法查看源代码的情况下查看类的成员。本实例将使用反射机制查看 `ArrayList` 类中定义的成员变量、构造方法和普通方法。实例的运行效果如图 7.17 所示。



Note

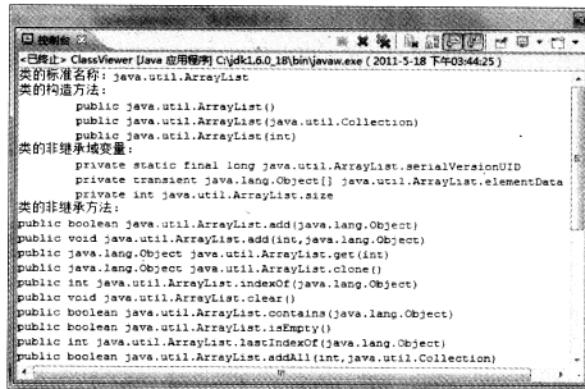


图 7.17 查看类的成员

实现过程

- (1) 在 Eclipse 中创建项目 072，并在该项目中创建 com.mingrisoft 包。
- (2) 在 com.mingrisoft 包中编写类 ClassViewer，在 main()方法中输出类中定义的构造方法、域和普通方法。关键代码如下：

```
public class ClassViewer {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) throws ClassNotFoundException {
        Class<?> clazz = Class.forName("java.util.ArrayList");
        System.out.println("类的标准名称: " + clazz.getCanonicalName());
        Constructor[] constructors = clazz.getConstructors(); //获得该类对象的所有构造方法
        System.out.println("类的构造方法: ");
        if (constructors.length != 0) {
            for (Constructor constructor : constructors) {
                System.out.println("\t" + constructor); //输出构造方法
            }
        } else {
            System.out.println("\t 空");
        }
        Field[] fields = clazz.getDeclaredFields(); //获得该类对象的所有非继承域
        System.out.println("类的非继承域变量: ");
        if (fields.length != 0) {
            for (Field field : fields) {
                System.out.println("\t" + field); //输出非继承域
            }
        } else {
            System.out.println("\t 空");
        }
        Method[] methods = clazz.getDeclaredMethods(); //获得该类对象的所有非继承方法
        System.out.println("类的非继承方法: ");
        if (methods.length != 0) {
            for (Method method : methods) {
                System.out.println(method); //输出非继承方法
            }
        }
    }
}
```



```
    } else {
        System.out.println("\t 空");
    }
}
```

技术要点

Note

Class 类的实例表示正在运行的 Java 应用程序中的类和接口。枚举是一种类，注释是一种接口。每个数组属于被映射为 Class 对象的一个类，所有具有相同元素类型和维数的数组都共享该 Class 对象。基本的 Java 类型（boolean、byte、char、short、int、long、float 和 double）和关键字 void 也表示为 Class 对象。它没有公共构造方法。Class 对象是在加载类时由 Java 虚拟机以及通过调用类加载器中的 defineClass() 方法自动构造的。本实例使用的方法如表 7.2 所示。

表 7.2 Class 类的常用方法

方法名	作用
getConstructors()	返回由该类对象的所有构造方法组成的数组
getDeclaredFields()	返回由该类对象的所有非继承域组成的数组
getDeclaredMethods()	返回由该类对象的所有非继承方法组成的数组

实例 073 查看内部类信息

(安裝位置： 配套資源\SL\07\073)

实例说明

Java 中支持在类的内部定义类，这种类称为内部类。内部类有些像 Java 中的方法，可以使用访问权限限定符修饰，也可以使用 static 关键字修饰等。本实例将利用 Java 的反射机制来查看内部类的信息。实例的运行效果如图 7.18 所示。

实现过程

(1) 在 Eclipse 中创建项目 073，并在该项目中创建 com.mingrisoft 包。

(2) 在 com.mingrisoft 包中编写类 NestedClassInformation，在该类的 main()方法中输出内部类的信息。关键代码如下：

```
public class NestedClassInformation {  
    public static void main(String[] args) throws ClassNotFoundException {  
        Class<?> cls = Class.forName("java.awt.geom.Point2D");  
        Class<?>[] classes = cls.getDeclaredClasses(); // 获得代表内部类的 Class 对象组成的数组  
        for (Class<?> clazz : classes) { // 遍历 Class 对象数组  
            System.out.println("类的标准名称: " + clazz.getCanonicalName());  
            System.out.println("类的修饰符: " + Modifier.toString(clazz.getModifiers()));  
            Type[] interfaces = clazz.getGenericInterfaces(); // 获得所有泛型接口
```

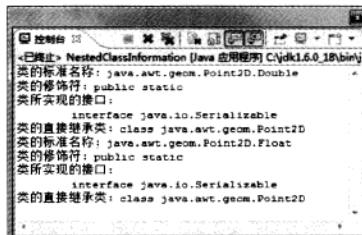


图 7.18 查看内部类信息



```
System.out.println("类所实现的接口: ");
if (interfaces.length != 0) { //如果泛型接口个数不是 0 则输出
    for (Type type : interfaces) {
        System.out.println("\t" + type);
    }
} else {
    System.out.println("\t" + "空");
}
Type superClass = clazz.getGenericSuperclass(); //获得直接父类
System.out.print("类的直接继承类: ");
if (superClass != null) { //如果直接父类不是 Object 就输出
    System.out.println(superClass);
} else {
    System.out.println("空");
}
}
```

技术要点

本实例主要应用 Class 类的 getDeclaredClasses()方法获得代表内部类的 Class 对象组成的数组。Class 类的 getDeclaredClasses()方法将返回 Class 对象的一个数组，这些对象包含声明为此 Class 对象所表示的类的成员的所有类和接口。包括该类所声明的公共、保护、默认（包）访问及私有类和接口，但不包括继承的类和接口。如果该类没有将任何类或接口声明为成员，或者此 Class 对象表示基本类型、数组类或 void，则此方法将返回一个长度为 0 的数组。该方法的声明如下：

```
public Class<?>[] getDeclaredClasses() throws SecurityException
```

对于私有的域或方法如果有安全管理器则可能会出现异常。

实例 074 动态设置类的私有域

(实例位置: 配套资源\SL\07\074)

实例说明

为了保证面向对象的封装特性，通常会将域设置成私有的，然后提供对应的 `getXXX()` 和 `setXXX()` 方法。对于非内部类而言，只能使用 `getXXX()` 和 `setXXX()` 方法来操作该域。然而利用反射机制，就可以在运行时修改类的私有域。本实例将通过简单的 `Student` 类来演示反射的这种用法。实例的运行效果如图 7.19 所示。



图 7.19 动态设置类的私有域

实现过程

- (1) 在 Eclipse 中创建项目 074，并在该项目中创建 com.mingrisoft 包。



(2) 在 com.mingrisoft 包中编写类 Student，在该类中定义 4 个域及其对应的 getXXX()和 setXXX()方法。这 4 个域分别是 id（表示学生的序号）、name（表示学生的姓名）、male（表示学生是否为男性）和 account（表示学生的账户余额）。关键代码如下：

```
public class Student {  
    private int id; //表示学生的序号  
    private String name; //表示学生的姓名  
    private boolean male; //表示学生的性别  
    private double account; //表示学生的账户余额  
    //省略了各个域的 getXXX() 和 setXXX() 方法  
}
```

(3) 在 com.mingrisoft 包中再编写类 Test 进行测试。在 main()方法中，分别为不同的域设置不同的值，并输出初始值和新值作为对比。关键代码如下：

```
public class Test {  
    public static void main(String[] args) {  
        Student student = new Student();  
        Class<?> clazz = student.getClass(); //获得代表 student 对象的 Class 对象  
        System.out.println("类的标准名称: " + clazz.getCanonicalName());  
        try {  
            Field id = clazz.getDeclaredField("id");  
            System.out.println("设置前的 id: " + student.getId());  
            id.setAccessible(true);  
            id.setInt(student, 10); //设置 id 值为 10  
            System.out.println("设置后的 id: " + student.getId());  
  
            Field name = clazz.getDeclaredField("name");  
            System.out.println("设置前的 name: " + student.getName());  
            name.setAccessible(true);  
            name.set(student, "明日科技"); //设置 name 值为明日科技  
            System.out.println("设置后的 name: " + student.getName());  
  
            Field male = clazz.getDeclaredField("male");  
            System.out.println("设置前的 male: " + student.isMale());  
            male.setAccessible(true);  
            male.setBoolean(student, true); //设置 male 值为 true  
            System.out.println("设置后的 male: " + student.isMale());  
  
            Field account = clazz.getDeclaredField("account");  
            System.out.println("设置前的 account: " + student.getAccount());  
            account.setAccessible(true);  
            account.setDouble(student, 12.34); //设置 account 值为 12.34  
            System.out.println("设置后的 account: " + student.getAccount());  
        } catch (SecurityException e) {  
            e.printStackTrace();  
        } catch (NoSuchFieldException e) {  
            e.printStackTrace();  
        } catch (IllegalArgumentException e) {  
            e.printStackTrace();  
        } catch (IllegalAccessException e) {  
            e.printStackTrace();  
        }  
    }  
}
```





```
        e.printStackTrace();
    }
}
```

技术要点

本实例主要应用了 Field 类的相关方法来实现动态设置类的私有域。Field 类提供有关类或接口的单个字段的信息，以及对它的动态访问权限。反射的字段可能是一个类（静态）字段或实例字段。本实例使用的方法如表 7.3 所示。



Note

表 7.3 Field 类的常用方法

方法名	作用
set(Object obj, Object value)	将指定对象变量上 Field 对象表示的字段设置为指定的新值
setBoolean(Object obj, boolean z)	将字段的值设置为指定对象上的一个 boolean 值
setDouble(Object obj, double d)	将字段的值设置为指定对象上的一个 double 值
setInt(Object obj, int i)	将字段的值设置为指定对象上的一个 int 值
setAccessible(boolean flag)	将此对象的 accessible 标志设置为指定的布尔值

脚下留神：

对于私有域，一定要使用 `setAccessible()`方法将其可见性设置为 `true` 才能设置新值。

实例 075 动态调用类中方法

(案例位置: 配套资源\SL\07\075)

实例说明

在 Java 中，调用类的方法有两种方式：对于静态方法可以直接使用类名调用，对于非静态方法必须使用类的对象调用。反射机制提供了比较另类的调用方式，可以根据需要指定要调用的方法，而不必在编程时确定。调用的方法不仅限于 public 的，还可以是 private 的。本实例将使用反射机制调用 Math 类的静态方法 sin() 和 String 类的非静态方法 equals()。实例的运行效果如图 7.20 所示。

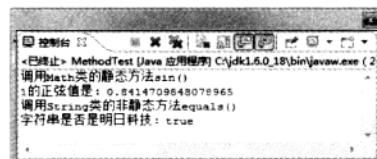


图 7.20 动态调用类中的方法

实现过程

- (1) 在 Eclipse 中创建项目 075，并在该项目中创建 com.mingrisoft 包。
 - (2) 在 com.mingrisoft 包中编写类 MethodTest，在该类的 main()方法中，分别调用了 Math 类的静态方法 sin() 和 String 类的非静态方法 equals()。关键代码如下：

```
public class MethodTest {  
    public static void main(String[] args) {  
        try {  
            System.out.println("调用 Math 类的静态方法 sin()");  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```



```
Method sin = Math.class.getDeclaredMethod("sin", Double.TYPE);
Double sin1 = (Double) sin.invoke(null, new Integer(1));
System.out.println("1 的正弦值是: " + sin1);
System.out.println("调用 String 类的非静态方法 equals()");
Method equals = String.class.getDeclaredMethod("equals", Object.class);
Boolean mrsoft = (Boolean) equals.invoke(new String("明日科技"), "明日科技");
System.out.println("字符串是否是明日科技: " + mrsoft);
} catch (Exception e) {
    e.printStackTrace();
}
}
```

指点迷津:

由于这几行代码会抛出大量异常，因此采用捕获 Exception 代替。通常不推荐这种写法。

技术要点

本实例主要通过 Method 类的相关方法实现。Method 类提供类或接口上单独某个方法（以及如何访问该方法）的信息。所反映的方法可能是类方法或实例方法（包括抽象方法）。它允许在匹配要调用的实参与底层方法的形参时进行扩展转换。但是如果要进行收缩转换，则会抛出异常 IllegalArgumentException。使用 Method 类的 invoke() 方法可以实现动态调用方法，该方法的声明如下：

```
public Object invoke(Object obj, Object... args) throws IllegalAccessException, IllegalArgumentException, InvocationTargetException
```

参数说明

- obj：从中调用底层方法的对象。
- args：用于方法调用的参数。

实例 076 动态实例化类

(实例位置：配套资源\SL\07\076)

实例说明

在 Java 中，通常是使用构造方法来创建对象的。构造方法可以分成有参数和无参数两种。如果类中没有定义构造方法，编译器会自动添加一个无参数的构造方法。使用构造方法创建对象虽然非常常用，但是并不灵活。本实例将演示如何使用反射创建对象。实例的运行效果如图 7.21 所示。

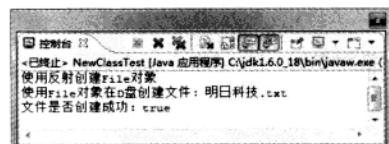


图 7.21 动态实例化类

实现过程

- (1) 在 Eclipse 中创建项目 076，并在该项目中创建 com.mingrisoft 包。
- (2) 在 com.mingrisoft 包中编写类 NewClassTest，在该类的 main() 方法中，创建 File 对象



并使用该对象在 D 盘创建一个文本文件。关键代码如下：

```
public class NewClassTest {
    public static void main(String[] args) {
        try {
            Constructor<File> constructor =
                File.class.getDeclaredConstructor(String.class);
            System.out.println("使用反射创建 File 对象");
            File file = constructor.newInstance("d://明日科技.txt");
            System.out.println("使用 File 对象在 D 盘创建文件：明日科技.txt");
            file.createNewFile(); // 创建新的文件
            System.out.println("文件是否创建成功：" + file.exists());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



Note

指点迷津：

由于这几行代码会抛出大量异常，因此采用捕获 Exception 代替。通常不推荐这种写法。

技术要点

本实例主要应用 Constructor 类及其相关方法实现。Constructor 类提供类的单个构造方法的信息以及对它的访问权限。它允许在将实参与带有底层构造方法的形参的 newInstance() 匹配时进行扩展转换，但是如果发生收缩转换，则抛出 IllegalArgumentException 异常。newInstance() 方法可以使用指定的参数来创建对象，该方法的声明如下：

```
public T newInstance(Object... initargs) throws InstantiationException, IllegalAccessException, InvocationTargetException
```

参数说明

initargs：作为变量传递给构造方法调用的对象数组。

实例 077 创建长度可变的数组

(实例位置：配套资源\SL\07\077)

实例说明

在 Java 中，对于数组的支持并不强大。程序员必须时刻注意数组中元素的个数，否则会出现数组下标越界异常。为此，在 Java API 中定义了 ArrayList 帮助开发，但这意味着需要学习新的方法。本实例将使用反射机制实现一个工具方法，每当调用该方法时数组的长度就会增加 5。实例的运行效果如图 7.22 所示。

实现过程

- (1) 在 Eclipse 中创建项目 077，并在该项目中创建 com.mingrisoft 包。

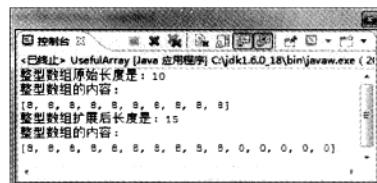


图 7.22 创建长度可变的数组



(2) 在 com.mingrisoft 包中编写类 UsefulArray。在该类中定义两个方法，一个是 increaseArray() 方法，用于将给定的 array 数组长度加 5；另一个是 main() 方法，用来进行测试。关键代码如下：

```
public class UsefulArray {
    public static Object increaseArray(Object array) {
        Class<?> clazz = array.getClass(); // 获得代表数组的 Class 对象
        if (clazz.isArray()) { // 如果输入是一个数组
            Class<?> componentType = clazz.getComponentType(); // 获得数组元素的类型
            int length = Array.getLength(array); // 获得输入的数组的长度
            Object newArray = Array.newInstance(componentType, length + 5); // 新建数组
            System.arraycopy(array, 0, newArray, 0, length); // 复制原来数组中的所有数据
            return newArray; // 返回新建数组
        }
        return null; // 如果输入的不是数组就返回空
    }

    public static void main(String[] args) {
        int[] intArray = new int[10];
        System.out.println("整型数组原始长度是：" + intArray.length);
        Arrays.fill(intArray, 8); // 将数组中的元素全部赋值为 8
        System.out.println("整型数组的内容：" );
        System.out.println(Arrays.toString(intArray));
        int[] newIntArray = (int[]) increaseArray(intArray); // 增加数组的长度
        System.out.println("整型数组扩展后长度是：" + newIntArray.length);
        System.out.println("整型数组的内容：" );
        System.out.println(Arrays.toString(newIntArray));
    }
}
```

指点迷津：

读者可以在本实例的基础上实现删除数组中的数据等方法。

技术要点

Array 类提供了动态创建和访问 Java 数组的方法。Array 允许在执行 getXXX() 或 setXXX() 方法操作期间进行扩展转换，但如果发生收缩转换，则抛出 IllegalArgumentException 异常。为了创建新的数组对象，需要使用 newInstance() 方法，它可以根据指定的元素类型和长度创建新的数组。该方法的声明如下：

```
public static Object newInstance(Class<?> componentType,int length) throws NegativeArraySizeException
```

参数说明

- componentType：表示新数组的组件类型的 Class 对象。
- length：新数组的长度。

为了获得给定数组的长度，需要使用 getLength() 方法，该方法的声明如下：

```
public static int getLength(Object array) throws IllegalArgumentException
```

参数说明

array：一个数组。

指点迷津：

对于 Java 的数组，不管几维，都属于 Object 类型。



实例 078 利用反射重写 toString()方法

(实例位置: 配套资源\SL\07\078)

实例说明

为了输出对象方便, Object 类提供了 `toString()` 方法。但是该方法的默认值是由类名和哈希码组成的, 实用性并不强。通常需要重写该方法以提供更多的信息。本实例主要实现重写类的 `toString()` 方法, 用于在调用 `toString()` 方法时, 使用反射输出类的包、类的名字、类的公共构造方法、类的公共域和类的公共方法。另外, 在重写该方法时, 为其传递一个 `Object` 型的参数, 这样可以避免多次重写 `toString()` 方法。实例的运行效果如图 7.23 所示。

```

<已停止> StringUtils [Java 应用程序] C:\jdk1.6.0_18\bin\javaw.exe (2011-5-18 下午04:51:41)
包名: java.util 类名: Date
公共构造方法:
public java.util.Date(long)
public java.util.Date(int,int,int)
public java.util.Date(int,int,int,int)
public java.util.Date(int,int,int,int,int)
public java.util.Date(java.lang.String)
public java.util.Date()
公共域:
公共方法:
public boolean java.util.Date.equals(java.lang.Object)
public java.lang.String java.util.Date.toString()
public int java.util.Date.hashCode()
public java.lang.Object java.util.Date.clone()
public int java.util.Date.compareTo(java.lang.Object)
public int java.util.Date.compareTo(java.util.Date)
public boolean java.util.Date.after(java.util.Date)
public boolean java.util.Date.before(java.util.Date)
public static long java.util.Date.parseLong(java.lang.String)
public int java.util.Date.getDate()
public long java.util.Date.getTime()
public void java.util.Date.setTime(long)

```

图 7.23 利用反射重写 `toString()` 方法

实现过程

- (1) 在 Eclipse 中创建项目 078, 并在该项目中创建 `com.mingrisoft` 包。
- (2) 在 `com.mingrisoft` 包中编写类 `StringUtils`。在该类中定义两个方法, 一个是 `toString()` 方法, 用于输出类的公共方法和域等信息; 另一个是 `main()` 方法, 用来进行测试。关键代码如下:

```

public class StringUtils {
    @SuppressWarnings("unchecked")
    public String toString(Object object) {
        Class clazz = object.getClass(); //获得代表该类的 Class 对象
        StringBuilder sb = new StringBuilder(); //利用 StringBuilder 来保存字符串
        Package packageName = clazz.getPackage(); //获得类所在的包
        sb.append("包名: " + packageName.getName() + "\t"); //输出类所在的包
        String className = clazz.getSimpleName(); //获得类的简单名称
        sb.append("类名: " + className + "\n"); //输出类的简单名称
        sb.append("公共构造方法: \n");
        //获得所有代表构造方法的 Constructor 数组
        Constructor[] constructors = clazz.getDeclaredConstructors();
        for (Constructor constructor : constructors) {

```



Note



```
String modifier = Modifier.toString(constructor.getModifiers()); //获得修饰符
if (modifier.contains("public")) { //查看修饰符是否含有“public”
    sb.append(constructor.toGenericString() + "\n");
}
}
sb.append("公共域: \n");
Field[] fields = clazz.getDeclaredFields(); //获得代表所有域的 Field 数组
for (Field field : fields) {
    String modifier = Modifier.toString(field.getModifiers());
    if (modifier.contains("public")) { //查看修饰符是否含有“public”
        sb.append(field.toGenericString() + "\n");
    }
}
sb.append("公共方法: \n");
Method[] methods = clazz.getDeclaredMethods(); //获得代表所有方法的 Method 数组
for (Method method : methods) {
    String modifier = Modifier.toString(method.getModifiers());
    if (modifier.contains("public")) { //查看修饰符是否含有“public”
        sb.append(method.toGenericString() + "\n");
    }
}
return sb.toString();
}
public static void main(String[] args) {
    System.out.println(new StringUtils().toString(new java.util.Date()));
}
}
```

技术要点

本实例应用到的技术与实例 072 介绍的相同，这里不再赘述。