# Глава 6
# ИНТЕРФЕЙСЫ И ВНУТРЕННИЕ КЛАССЫ

## Интерфейсы

Интерфейсы подобны полностью абстрактным классам, но не являются классами. Ни один из объявленных методов не может быть реализован внутри интерфейса. В языке Java существуют два вида интерфейсов: интерфейсы, определяющие контракт для классов посредством методов, и интерфейсы, реализация которых автоматически (без реализации методов) придает классу определенные свойства. К последним относятся, например, интерфейсы **Cloneable** и **Serializable**, отвечающие за клонирование и сохранение объекта в информационном потоке соответственно.

Все объявленные в интерфейсе методы автоматически трактуются как **public** и **abstract**, а все поля – как **public**, **static** и **final**, даже если они так не объявлены. Класс может реализовывать любое число интерфейсов, указываемых через запятую после ключевого слова **implements**, дополняющего определение класса. После этого класс обязан реализовать все методы, полученные им от интерфейсов, или объявить себя абстрактным классом.

На множестве интерфейсов также определена иерархия наследования, но она не имеет отношения к иерархии классов.

Определение интерфейса имеет вид:

```
[public] interface Имя [extends Имя1, Имя2,…, ИмяN] {
                                        /*реализация интерфейса*/}
```

Например:

```
/* пример # 1 : объявление интерфейсов: LineGroup.java, Shape.java */
package chapt06;

public interface LineGroup {
        // по умолчанию public abstract
        double getPerimeter();// объявление метода
}

package chapt06;

public interface Shape extends LineGroup {
      //int id; // ошибка, если нет инициализации
      //void method(){} /* ошибка, так как абстрактный метод не может
                          иметь тела!  */
        double getSquare();  // объявление метода
}
```

Для более простой идентификации интерфейсов в большом проекте в сообществе разработчиков действует негласное соглашение о добавлении к имени интерфейса символа **'I'**, в соответствии с которым вместо имени **Shape** можно записать **IShape**.

Класс, который будет реализовывать интерфейс **Shape**, должен будет определить все методы из цепочки наследования интерфейсов. В данном случае это методы **getPerimeter()** и **getSquare()**.

Интерфейсы обычно объявляются как **public**, потому что описание функциональности, предоставляемое ими, может быть использовано в нескольких пакетах проекта. Интерфейсы с областью видимости в рамках пакета могут использоваться только в этом пакете и нигде более.

В языке Java интерфейсы обеспечивают большую часть той функциональности, которая в C++ представляется с помощью механизма множественного наследования. Класс может наследовать один суперкласс и реализовывать произвольное число интерфейсов.

Реализация интерфейсов классом может иметь вид:

```
[доступ] class ИмяКласса implements Имя1, Имя2,…, ИмяN {
                                        /*код класса*/}
```

Здесь **Имя1, Имя2,…, ИмяN** – перечень используемых интерфейсов. Класс, который реализует интерфейс, должен предоставить полную реализацию всех методов, объявленных в интерфейсе. Кроме этого, данный класс может объявлять свои собственные методы. Если класс расширяет интерфейс, но полностью не реализует его методы, то этот класс должен быть объявлен как **abstract.**

```
/* пример # 2 : реализация интерфейса: Rectangle.java */
package chapt06;

public class Rectangle implements Shape {
      private double a, b;

      public Rectangle(double a, double b) {
            this.a = a;
            this.b = b;
      }
      //реализация метода из интерфейса
      public double getSquare() {//площадь прямоугольника
            return a * b;
      }
      //реализация метода из интерфейса
      public double getPerimeter() {
            return 2 * (a + b);
      }
}

/* пример # 3 : реализация интерфейса: Circle.java */
package chapt06;

public class Circle implements Shape {
```

```
      private double r;

      public Circle(double r) {
            this.r = r;
      }
      public double getSquare() {//площадь круга
            return Math.PI * Math.pow(r, 2);
      }
      public double getPerimeter() {
            return 2 * Math.PI * r;
      }
}
```

```
/* пример # 4 : неполная реализация интерфейса: Triangle.java */
package chapt06;
/* метод getSquare() в данном абстрактном классе не реализован */
public abstract class Triangle implements Shape {
      private double a, b, c;

      public Triangle(double a, double b, double c) {
            this.a = a;
            this.b = b;
            this.c = c;
      }
      public double getPerimeter() {
            return a + b + c;
      }
}
```

```
/* пример # 5 : свойства ссылки на интерфейс : Runner.java */
package chapt06;

public class Runner {
      public static void printFeatures(Shape f) {
         System.out.printf("площадь:%.2f периметр: %.2f%n",
                  f.getSquare(), f.getPerimeter());
      }
      public static void main(String[] args) {
            Rectangle r = new Rectangle(5, 9.95);
            Circle c = new Circle(7.01);
            printFeatures(r);
            printFeatures(c);
      }
}
```

В результате будет выведено:

```
площадь:49,75 периметр: 29,90
площадь:154,38 периметр: 44,05
```

Класс **Runner** содержит метод **printFeatures(Shape f)**, который вызывает методы объекта, передаваемого ему в качестве параметра. Вначале ему передается объект, соответствующий прямоугольнику, затем кругу (объекты **c**

и **r**). Каким образом метод **printFeatures()** может обрабатывать объекты двух различных классов? Все дело в типе передаваемого этому методу аргумента – класса, реализующего интерфейс **Shape**. Вызывать, однако, можно только те методы, которые были объявлены в интерфейсе.

В следующем примере в классе **ShapeCreator** используются классы и интерфейсы, определенные выше, и объявляется ссылка на интерфейсный тип. Такая ссылка может указывать на экземпляр любого класса, который реализует объявленный интерфейс. При вызове метода через такую ссылку будет вызываться его реализованная версия, основанная на текущем экземпляре класса. Выполняемый метод разыскивается динамически во время выполнения, что позволяет создавать классы позже кода, который вызывает их методы.

```
/* пример # 6 : динамический связывание методов : ShapeCreator.java */
package chapt06;

public class ShapeCreator {
      public static void main(String[] args) {
            Shape sh; /* ссылка на интерфейсный тип */
            Rectangle re = new Rectangle(5, 9.95);
            sh = re;
            sh.getPerimeter();//вызов метода класса Rectangle
            Circle cr = new Circle(7.01);
            sh = cr; // присваивается ссылка на другой объект
            sh.getPerimeter();//вызов метода класса Circle

            // cr=re; // ошибка! разные ветви наследования
      }
}
```

Невозможно приравнивать ссылки на классы, находящиеся в разных ветвях наследования, так как не существует никакого способа привести один такой тип к другому. По этой же причине ошибку вызовет попытка объявления объекта в виде:

```
Circle c = new Rectangle(1, 5);
```

## Пакеты

Любой класс Java относится к определенному пакету, который может быть неименованным (unnamed или default package), если оператор **package** отсутствует. Оператор **package** имя, помещаемый в начале исходного программного файла, определяет именованный пакет, т.е. область в пространстве имен классов, где определяются имена классов, содержащихся в этом файле. Действие оператора **package** указывает на месторасположение файла относительно корневого каталога проекта. Например:

```
package chapt06;
```

При этом программный файл будет помещен в подкаталог с названием **chapt06**. Имя пакета при обращении к классу из другого пакета присоединяется к имени класса: **chapt06.Student**. Внутри указанной области можно выделить подобласти:

```
package chapt06.bsu;
```