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;
```

Общая форма файла, содержащего исходный код Java, может быть следующая:

```
одиночный оператор package (необязателен);
любое количество операторов import (необязательны);
одиночный открытый (public) класс (необязателен)
любое количество классов пакета (необязательны)
```

В реальных проектах пакеты часто именуются следующим образом:

- обратный интернет-адрес производителя программного обеспечения, а именно для `www.bsu.by` получится `by.bsu`;
- далее следует имя проекта, например: `eun`;
- затем располагаются пакеты, определяющие собственно приложение.

При использовании классов перед именем класса через точку надо добавлять полное имя пакета, к которому относится данный класс. На рисунке приведен далеко не полный список пакетов реального приложения. Из названий пакетов можно определить, какие примерно классы в нем расположены, не заглядывая внутрь. При создании пакета всегда следует руководствоваться простым правилом: называть его именем простым, но отражающим смысл, логику поведения и функциональность объединенных в нем классов.

by.bsu.eun
by.bsu.eun.administration.constants
by.bsu.eun.administration.dbhelpers
by.bsu.eun.common.constants
by.bsu.eun.common.dbhelpers.annboard
by.bsu.eun.common.dbhelpers.courses
by.bsu.eun.common.dbhelpers.guestbook
by.bsu.eun.common.dbhelpers.learnres
by.bsu.eun.common.dbhelpers.messages
by.bsu.eun.common.dbhelpers.news
by.bsu.eun.common.dbhelpers.prepinfo
by.bsu.eun.common.dbhelpers.statistics
by.bsu.eun.common.dbhelpers.subjectmark
by.bsu.eun.common.dbhelpers.subjects
by.bsu.eun.common.dbhelpers.test
by.bsu.eun.common.dbhelpers.users
by.bsu.eun.common.menus
by.bsu.eun.common.objects
by.bsu.eun.common.servlets
by.bsu.eun.common.tools
by.bsu.eun.consultation.constants
by.bsu.eun.consultation.dbhelpers
by.bsu.eun.consultation.objects
by.bsu.eun.core.constants
by.bsu.eun.core.dbhelpers
by.bsu.eun.core.exceptions
by.bsu.eun.core.filters
by.bsu.eun.core.managers
by.bsu.eun.core.taglibs

Рис. 6.1. Организация пакетов приложения

Каждый класс добавляется в указанный пакет при компиляции. Например:

```
// пример # 7 : простейший класс в пакете: CommonObject.java
package by.bsu.eun.objects;

public class CommonObject implements Cloneable {
      public CommonObject() {
            super();
      }
      public Object clone()
            throws CloneNotSupportedException {
            return super.clone();
      }
}
```

Класс начинается с указания того, что он принадлежит пакету **by.bsu.eun.objects**. Другими словами, это означает, что файл **CommonObject.java** находится в каталоге **objects**, который, в свою очередь, находится в каталоге **bsu**, и так далее. Нельзя переименовывать пакет, не переименовав каталог, в котором хранятся его классы. Чтобы получить доступ к классу из другого пакета, перед именем такого класса указывается имя пакета: **by.bsu.eun.objects.CommonObject**. Чтобы избежать таких длинных имен, используется ключевое слово **import.** Например:

```
import by.bsu.eun.objects.CommonObject;
```

или

```
import by.bsu.eun.objects.*;
```

Во втором варианте импортируется весь пакет, что означает возможность доступа к любому классу пакета, но только не к подпакету и его классам. В практическом программировании следует использовать индивидуальный **import** класса, чтобы при анализе кода была возможность быстро определить месторасположение используемого класса.

Доступ к классу из другого пакета можно осуществить следующим образом:

```
// пример # 8 : доступ к пакету: UserStatistic.java
package by.bsu.eun.usermng;

public class UserStatistic
                  extends by.bsu.eun.objects.CommonObject {
      private long id;
      private int mark ;

      public UserStatistic() {
            super();
      }
      public long getId() {
            return id;
      }
      public void setId(long id) {
```

```
            this.id = id;
      }
      public int getMark() {
            return mark;
      }
      public void setMark(int mark) {
            this.mark = mark;
      }
}
```

При импорте класса из другого пакета рекомендуется всегда указывать полный путь с указанием имени импортируемого класса. Это позволяет в большом проекте легко найти определение класса, если возникает необходимость посмотреть исходный код класса.

```
// пример # 9 : доступ к пакету: CreatorStatistic.java
package by.bsu.eun.actions;
import by.bsu.eun.objects.CommonObject;
import by.bsu.eun.usermng.UserStatistic;

public class CreatorStatistic {
   public static UserStatistic createUserStatistic(long id)
{
            UserStatistic temp = new UserStatistic();
            temp.setId(id);
            // чтение информации из базы данных по id пользователя
            int mark = полученное значение;
            temp.setMark(mark);
            return temp;
      }
      public static void main(String[] args) {
            UserStatistic us = createUserStatistic(71);
            System.out.println(us.getMark());
      }
}
```

Если пакет не существует, то его необходимо создать до первой компиляции, если пакет не указан, класс добавляется в пакет без имени (unnamed). При этом unnamed-каталог не создается. Однако в реальных проектах классы вне пакетов не создаются, и не существует причин отступать от этого правила.

Статический импорт

Константы и статические методы класса можно использовать без указания принадлежности к классу, если применить статический импорт, как это показано в следующем примере.

```
// пример # 10 : статический импорт: ImportDemo.java
package chapt06;
import static java.lang.Math.*;

public class ImportDemo {
```

```
    public static void main(String[] args) {
        double radius = 3;
        System.out.println(2 * PI * radius);
        System.out.println(floor(cos(PI/3)));
    }
}
```

Если необходимо получить доступ только к одной константе класса или интерфейса, например **Math.E**, то статический импорт производится в следующем виде:

```
import static java.lang.Math.E;
import static java.lang.Math.cos;//для одного метода
```

Внутренние классы

Классы могут взаимодействовать друг с другом не только посредством наследования и использования ссылок, но и посредством организации логической структуры с определением одного класса в теле другого.

В Java можно определить (вложить) один класс внутри определения другого класса, что позволяет группировать классы, логически связанные друг с другом, и динамично управлять доступом к ним. С одной стороны, обоснованное использование в коде внутренних классов делает его более эффектным и понятным. С другой стороны, применение внутренних классов есть один из способов сокрытия кода, так как внутренний класс может быть абсолютно недоступен и не виден вне класса-владельца. Внутренние классы также используются в качестве блоков прослушивания событий (глава «События»). Одной из важнейших причин использования внутренних классов является возможность независимого наследования внутренними классами. Фактически при этом реализуется множественное наследование со своими преимуществами и проблемами.

В качестве примеров можно рассмотреть взаимосвязи классов «Корабль», «Двигатель» и «Шлюпка». Объект класса «Двигатель» расположен внутри (невидим извне) объекта «Корабль» и его деятельность приводит «Корабль» в движение. Оба этих объекта неразрывно связаны, то есть запустить «Двигатель» можно только посредством использования объекта «Корабль», например, из машинного отделения. Таким образом, перед инициализацией объекта внутреннего класса «Двигатель» должен быть создан объект внешнего класса «Корабль».

Класс «Шлюпка» также является логической частью класса «Корабль», однако ситуация с его объектами проще по причине того, что данные объекты могут быть использованы независимо от наличия объекта «Корабль». Объект класса «Шлюпка» только использует имя (на борту) своего внешнего класса. Такой внутренний класс следует определять как **static**. Если объект «Шлюпка» используется без привязки к какому-либо судну, то класс следует определять как обычный независимый класс.

Вложенные классы могут быть статическими, объявляемыми с модификатором **static**, и нестатическими. Статические классы могут обращаться к членам включающего класса не напрямую, а только через его объект. Нестатические внутренние классы имеют доступ ко всем переменным и методам своего внешнего класса-владельца.

Внутренние (inner) классы

Нестатические вложенные классы принято называть внутренними (inner) классами. Доступ к элементам внутреннего класса возможен из внешнего класса только через объект внутреннего класса, который должен быть создан в коде метода внешнего класса. Объект внутреннего класса всегда ассоциируется (скрыто хранит ссылку) с создавшим его объектом внешнего класса – так называемым внешним (enclosing) объектом. Внешний и внутренний классы могут выглядеть, например, так:

```
public class Ship {
      // поля и конструкторы
      // abstract, final, private, protected - допустимы
      public class Engine {  // определение внутреннего класса
            // поля и методы
            public void launch() {
               System.out.println("Запуск двигателя");
            }
      }// конец объявления внутреннего класса
      public void init() {// метод внешнего класса
        // объявление объекта внутреннего класса
            Engine eng = new Engine();
            eng.launch();
      }
}
```

При таком объявлении объекта внутреннего класса **Engine** в методе внешнего класса **Ship** нет реального отличия от использования какого-либо другого внешнего класса, кроме объявления внутри класса **Ship**. Использование объекта внутреннего класса вне своего внешнего класса возможно только при наличии доступа (видимости) и при объявлении ссылки в виде:

```
Ship.Engine obj = new Ship().new Engine();
```

Основное отличие от внешнего класса состоит в больших возможностях ограничения видимости внутреннего класса по сравнению с обычным внешним классом. Внутренний класс может быть объявлен как **private**, что обеспечивает его полную невидимость вне класса-владельца и надежное сокрытие реализации. В этом случае ссылку **obj**, приведенную выше, объявить было бы нельзя. Создать объект такого класса можно только в методах и логических блоках внешнего класса. Использование **protected** позволяет получить доступ к внутреннему классу для класса в другом пакете, являющегося суперклассом внешнего класса.

После компиляции объектный модуль, соответствующий внутреннему классу, получит имя **Ship$Engine.class**.

Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса, в то же время внешний класс может получить доступ к содержимому внутреннего класса только после создания объекта внутреннего класса. Внутренние классы не могут содержать статические атрибуты и методы, кроме констант (**final static**). Внутренние классы имеют право наследовать другие классы, реализовывать интерфейсы и выступать в роли объектов наследования. Допустимо наследование следующего вида:

```
public class WarShip extends Ship {
     protected class SpecialEngine extends Engine {}
}
```

Если внутренний класс наследуется обычным образом другим классом (после **extends** указывается **ИмяВнешнегоКласса.ИмяВнутреннегоКласса**), то он теряет доступ к полям своего внешнего класса, в котором он был объявлен.

```
public class Motor extends Ship.Engine {
     public Motor(Ship obj) {
          obj.super();
     }
}
```

В данном случае конструктор класса **Motor** должен быть объявлен с параметром типа **Ship**, что позволит получить доступ к ссылке на внутренний класс **Engine**, наследуемый классом **Motor**.

Внутренние классы позволяют окончательно решить проблему множественного наследования, когда требуется наследовать свойства нескольких классов.

При объявлении внутреннего класса могут использоваться модификаторы **final**, **abstract**, **private**, **protected**, **public**.

Простой пример практического применения взаимодействия класса-владельца и внутреннего нестатического класса проиллюстрирован на следующем примере.

```
/* пример # 11 : взаимодействие внешнего и внутреннего классов : Student.java :
AnySession.java */
package chapt06;

public class Student {
     private int id;
     private ExamResult[] exams;

     public Student(int id) {
          this.id = id;
     }

     private class ExamResult {// внутренний класс
          private String name;
          private int mark;
          private boolean passed;

          public ExamResult(String name) {
               this.name = name;
               passed = false;
          }
          public void passExam() {
               passed = true;
          }
          public void setMark(int mark) {
               this.mark = mark;
          }
```

```
        public int getMark() {
             return mark;
        }
        public int getPassedMark() {
             final int PASSED_MARK = 4;// «волшебное» число
             return PASSED_MARK;
        }
        public String getName() {
             return name;
        }
        public boolean isPassed() {
             return passed;
        }
    } // окончание внутреннего класса

public void setExams(String[] name, int[] marks) {
            if (name.length != marks.length)
                  throw new IllegalArgumentException();
            exams = new ExamResult[name.length];
            for (int i = 0; i < name.length; i++) {
                  exams[i] = new ExamResult(name[i]);
                  exams[i].setMark(marks[i]);
      if (exams[i].getMark() >= exams[i].getPassedMark())
                        exams[i].passExam();
            }
public String toString() {
            String res = "Студент: " + id + "\n";
      for (int i = 0; i < exams.length; i++)
            if (exams[i].isPassed())
            res += exams[i].getName() + " сдал \n";
      else
            res += exams[i].getName() + " не сдал \n";

      return res;
      }
}
package chapt06;

public class AnySession {
     public static void main(String[] args) {
          Student stud = new Student(822201);
          String ex[] = {"Механика","Программирование"};
          int marks[] = { 2, 9 };
          stud.setExams(ex, marks);
          System.out.println(stud);
     }
}
```

В результате будет выведено:

```
Студент: 822201
Механика не сдал
Программирование сдал
```

Внутренний класс определяет сущность предметной области "результат экзамена" (класс **ExamResult**), которая обычно непосредственно связана в информационной системе с объектом класса **Student**. Класс **ExamResult** в данном случае определяет только методы доступа к своим атрибутам и совершенно невидим вне класса **Student**, который включает методы по созданию и инициализации массива объектов внутреннего класса с любым количеством экзаменов, который однозначно идентифицирует текущую успеваемость студента.

Внутренний класс может быть объявлен также внутри метода или логического блока внешнего класса. Видимость такого класса регулируется областью видимости блока, в котором он объявлен. Но внутренний класс сохраняет доступ ко всем полям и методам внешнего класса, а также ко всем константам, объявленным в текущем блоке кода. Класс, объявленный внутри метода, не может быть объявлен как **static**, а также не может содержать статические поля и методы.

```
/*пример # 12 : внутренний класс, объявленный внутри метода:
TeacherLogic.java*/
package chapt06;

public abstract class AbstractTeacher {
      private int id;
      public AbstractTeacher(int id) {
            this.id = id;
      }
      public abstract boolean excludeStudent(String name);
}
package chapt06;

public class Teacher extends AbstractTeacher {

      public Teacher(int id) {
            super(id);
      }
      public boolean excludeStudent(String name) {
            return false;
      }
}
package chapt06;

public class TeacherCreator {
      public TeacherCreator(){}

      public AbstractTeacher createTeacher(int id) {
            // объявление класса внутри метода
         class Dean extends AbstractTeacher {
```

```
                Dean(int id){
                      super(id);
                }
          public boolean excludeStudent(String name) {
                      if (name != null) {
                      // изменение статуса студента в базе данных
                            return true;
                      }
                      else return false;
                }
          }// конец внутреннего класса

          if (isDeanId(id))
                return new Dean(id);
          else return new Teacher(id);
    }
    private static boolean isDeanId(int id) {
          // проверка декана из БД или
          return (id  == 777);
    }
}
package chapt06;

public class TeacherLogic {
    public static void excludeProcess(int deanId,
                String name) {
          AbstractTeacher teacher =
          new TeacherCreator().createTeacher(deanId);

          System.out.println("Студент: " + name
          + " отчислен:" + teacher.excludeStudent(name));
    }
    public static void main(String[] args) {
          excludeProcess(700, "Балаганов");
          excludeProcess(777, "Балаганов");
    }
}
```

В результате будет выведено:

```
Студент: Балаганов отчислен:false
Студент: Балаганов отчислен:true
```

Класс **Dean** объявлен в методе **createTeacher(int id)**, и соответственно объекты этого класса можно создавать только внутри этого метода, из любого другого места внешнего класса внутренний класс недоступен. Однако существует возможность получить ссылку на класс, объявленный внутри метода, и использовать его специфические свойства. При компиляции данного кода с внутренним классом ассоциируется объектный модуль со сложным именем **TeacherCreator$1Dean**, тем не менее однозначно определяющим связь между внешним и внутренним классами. Цифра **1** в имени говорит о том, что в других методах класса могут быть объявлены внутренние классы с таким же именем.

Вложенные (nested) классы

Если не существует необходимости в связи объекта внутреннего класса с объектом внешнего класса, то есть смысл сделать такой класс статическим.

Вложенный класс логически связан с классом-владельцем, но может быть использован независимо от него.

При объявлении такого внутреннего класса присутствует служебное слово **static**, и такой класс называется вложенным (nested). Если класс вложен в интерфейс, то он становится статическим по умолчанию. Такой класс способен наследовать другие классы, реализовывать интерфейсы и являться объектом наследования для любого класса, обладающего необходимыми правами доступа. В то же время статический вложенный класс для доступа к нестатическим членам и методам внешнего класса должен создавать объект внешнего класса, а напрямую имеет доступ только к статическим полям и методам внешнего класса. Для создания объекта вложенного класса объект внешнего класса создавать нет необходимости. Подкласс вложенного класса не способен унаследовать возможность доступа к членам внешнего класса, которыми наделен его суперкласс.

```
/* пример # 13 : вложенный класс: Ship.java : RunnerShip.java */
package chapt06;

public class Ship {
      private int id;
      // abstract, final, private, protected - допустимы
      public static class LifeBoat {
            public static void down() {
                  System.out.println("шлюпки на воду!");
            }
            public void swim() {
                  System.out.println("отплытие шлюпки");
            }
      }
}
package chapt06;

public class RunnerShip {
      public static void main(String[] args) {
            // вызов статического метода
            Ship.LifeBoat.down();
            // создание объекта статического класса
            Ship.LifeBoat lf = new Ship.LifeBoat();
            // вызов обычного метода
            lf.swim();
      }
}
```

Статический метод вложенного класса вызывается при указании полного относительного пути к нему. Объект **lf** вложенного класса создается с использованием имени внешнего класса без вызова его конструктора.

Класс, вложенный в интерфейс, по умолчанию статический. На него не накладывается никаких особых ограничений, и он может содержать поля и методы как статические, так и нестатические.

```
/* пример # 14 : класс вложенный в интерфейс: Faculty.java : University.java */
package chapt06;

public interface University {
      int NUMBER_FACULTY = 20;

      class LearningDepartment {// static по умолчанию
            public int idChief;

            public static void assignPlan(int idFaculty) {
                  // реализация
            }
            public void acceptProgram() {
                  // реализация
            }
      }
}
```

Такой внутренний класс использует пространство имен интерфейса.

Анонимные (anonymous) классы

Анонимные (безымянные) классы применяются для придания уникальной функциональности отдельно взятому объекту для обработки событий, реализации блоков прослушивания и т.д. Можно объявить анонимный класс, который будет расширять другой класс или реализовывать интерфейс при объявлении одного, единственного объекта, когда остальным объектам этого класса будет соответствовать реализация метода, определенная в самом классе. Объявление анонимного класса выполняется одновременно с созданием его объекта посредством оператора **new**.

Анонимные классы эффективно используются, как правило, для реализации (переопределения) нескольких методов и создания собственных методов объекта. Этот прием эффективен в случае, когда необходимо переопределение метода, но создавать новый класс нет необходимости из-за узкой области (или одноразового) применения метода.

Конструкторы анонимных классов нельзя определять и переопределять. Анонимные классы допускают вложенность друг в друга, что может сильно запутать код и сделать эти конструкции непонятными.

```
/* пример # 15 : анонимные классы: TypeQuest.java: RunnerAnonym.java */
package chapt06;

public class TypeQuest {
      private int id = 1;

      public TypeQuest() {
      }
```

```
      public TypeQuest(int id) {
            this.id = id;
      }
      public void addNewType() {
            // реализация
            System.out.println(
            "добавлен вопрос на соответствие");
      }
}
package chapt06;

public class RunnerAnonym {
      public static void main(String[] args) {
       TypeQuest unique = new TypeQuest() {// анонимный класс #1
                  public void addNewType() {
                        // новая реализация метода
                        System.out.println(
                  "добавлен вопрос со свободным ответом");
                  }
            };// конец объявления анонимного класса
            unique.addNewType();

            new TypeQuest(71) {// анонимный класс #2
                  private String name = "Drag&Drop";

                  public void addNewType() {
                        // новая реализация метода #2
            System.out.println("добавлен " + getName());
                  }
                  String getName() {
                        return name;
                  }
            }.addNewType();

            TypeQuest standard = new TypeQuest(35);
            standard.addNewType();
      }
}
```

В результате будет выведено:

```
добавлен вопрос со свободным ответом
добавлен Drag&Drop
добавлен вопрос на соответствие
```

При запуске приложения происходит объявление объекта **unique** с применением анонимного класса, в котором переопределяется метод **addNewType()**. Вызов данного метода на объекте **unique** приводит к вызову версии метода из анонимного класса, который компилируется в объектный модуль с именем **RunnerAnonym$1**. Процесс создания второго объекта с анонимным типом применяется в программировании значительно чаще, особенно при реализации клас-

сов-адаптеров и реализации интерфейсов в блоках прослушивания. В этом же объявлении продемонстрирована возможность объявления в анонимном классе полей и методов, которые доступны объекту вне этого класса.

Для перечисления объявление анонимного внутреннего класса выглядит несколько иначе, так как инициализация всех элементов происходит при первом обращении к типу. Поэтому и анонимный класс реализуется только внутри объявления типа **enum**, как это сделано в следующем примере.

```
/* пример # 16 : анонимный класс в перечислении : EnumRunner.java */
package chapt06;

enum Shape {
      RECTANGLE, SQUARE,
      TRIANGLE {// анонимный класс
            public double getSquare() {// версия для TRIANGLE
                  return a*b/2;
            }
      };
      public double a, b;

      public void setShape(double a, double b){
            if ((a<=0 || b<=0) || a!=b && this==SQUARE)
                  throw new IllegalArgumentException();
            else
                  this.a = a;
                  this.b = b;
      }
      public double getSquare(){// версия для RECTANGLE и SQUARE
            return a * b;
      }
      public String getParameters() {
            return "a=" + a + ", b=" + b;
      }
}
public class EnumRunner {
      public static void main(String[] args) {
            int i = 4;
            for (Shape f : Shape.values()) {
                  f.setShape(3, i--);
      System.out.println(f.name()+"-> " + f.getParameters()
                        + " площадь= " + f.getSquare());
                  }
      }
}
```

В результате будет выведено:

```
RECTANGLE-> a=3.0, b=4.0 площадь= 12.0
SQUARE-> a=3.0, b=3.0 площадь= 9.0
TRIANGLE-> a=3.0, b=2.0 площадь= 3.0
```

Объектный модуль для такого анонимного класса будет скомпилирован с именем **Shape$1**.

Задания к главе 6

Вариант А

1. Создать класс **Notepad** (записная книжка) с внутренним классом или классами, с помощью объектов которого могут храниться несколько записей на одну дату.
2. Создать класс **Payment** (покупка) с внутренним классом, с помощью объектов которого можно сформировать покупку из нескольких товаров.
3. Создать класс **Account** (счет) с внутренним классом, с помощью объектов которого можно хранить информацию обо всех операциях со счетом (снятие, платежи, поступления).
4. Создать класс **Зачетная Книжка** с внутренним классом, с помощью объектов которого можно хранить информацию о сессиях, зачетах, экзаменах.
5. Создать класс **Department** (отдел фирмы) с внутренним классом, с помощью объектов которого можно хранить информацию обо всех должностях отдела и обо всех сотрудниках, когда-либо занимавших конкретную должность.
6. Создать класс **Catalog** (каталог) с внутренним классом, с помощью объектов которого можно хранить информацию об истории выдач книги читателям.
7. Создать класс **СССР** с внутренним классом, с помощью объектов которого можно хранить информацию об истории изменения территориального деления на области и республики.
8. Создать класс **City** (город) с внутренним классом, с помощью объектов которого можно хранить информацию о проспектах, улицах, площадях.
9. Создать класс **CD** (mp3-диск) с внутренним классом, с помощью объектов которого можно хранить информацию о каталогах, подкаталогах и записях.
10. Создать класс **Mobile** с внутренним классом, с помощью объектов которого можно хранить информацию о моделях телефонов и их свойствах.
11. Создать класс **Художественная Выставка** с внутренним классом, с помощью объектов которого можно хранить информацию о картинах, авторах и времени проведения выставок.
12. Создать класс **Календарь** с внутренним классом, с помощью объектов которого можно хранить информацию о выходных и праздничных днях.
13. Создать класс **Shop** (магазин) с внутренним классом, с помощью объектов которого можно хранить информацию об отделах, товарах и услуг.
14. Создать класс **Справочная Служба Общественного Транспорта** с внутренним классом, с помощью объектов которого можно хранить информацию о времени, линиях маршрутов и стоимости проезда.
15. Создать класс **Computer** (компьютер) с внутренним классом, с помощью объектов которого можно хранить информацию об операционной системе, процессоре и оперативной памяти.

16. Создать класс **Park** (парк) с внутренним классом, с помощью объектов которого можно хранить информацию об аттракционах, времени их работы и стоимости.
17. Создать класс **Cinema** (кино) с внутренним классом, с помощью объектов которого можно хранить информацию об адресах кинотеатров, фильмах и времени сеансов.
18. Создать класс **Программа Передач** с внутренним классом, с помощью объектов которого можно хранить информацию о названии телеканалов и программ.
19. Создать класс **Фильм** с внутренним классом, с помощью объектов которого можно хранить информацию о продолжительности, жанре и режиссерах фильма.

Вариант B

В заданиях варианта B главы 4 в одном из классов для сокрытия реализации использовать внутренний или вложенный класс. Для определения уникального поведения объекта одного из классов использовать анонимные классы.

Вариант C

Реализовать абстрактные классы или интерфейсы, а также наследование и полиморфизм для следующих классов:

1. Абстрактный класс **Книга** (Шифр, Автор, Название, Год, Издательство). Подклассы **Справочник** и **Энциклопедия**.
2. **`interface` Абитуриент ← `abstract class` Студент ← `class` Студент-Заочник**.
3. **`interface` Сотрудник ← `class` Инженер ← `class` Руководитель**.
4. **`interface` Здание ← `abstract class` Общественное Здание ← `class` Театр**.
5. **`interface` Mobile ← `abstract class` Siemens Mobile ← `class` Model**.
6. **`interface` Корабль ← `abstract class` Военный Корабль ← `class` Авианосец**.
7. **`interface` Врач ← `class` Хирург ← `class` Нейрохирург**.
8. **`interface` Корабль ← `class` Грузовой Корабль ← `class` Танкер**.
9. **`interface` Мебель ← `abstract class` Шкаф ← `class` Книжный Шкаф**.
10. **`interface` Фильм ← `class` Отечественный Фильм ← `class` Комедия**.
11. **`interface` Ткань ← `abstract class` Одежда ← `class` Костюм**.
12. **`interface` Техника ← `abstract class` Плеер ← `class` Видеоплеер**.
13. **`interface` Транспортное Средство ← `abstract class` Общественный Транспорт ← `class` Трамвай**.

14. **interface Устройство Печати ← class Принтер ← class Лазерный Принтер**.
15. **interface Бумага ← abstract class Тетрадь ← class Тетрадь Для Рисования**.
16. **interface Источник Света ← class Лампа ← class Настольная Лампа**.

Тестовые задания к главе 6

Вопрос 6.1.

Какие из фрагментов кода скомпилируются без ошибки?

1)

```
import java.util.*;
package First;
class My{/* тело класса*/}
```

2)

```
package mypack;
import java.util.*;
public class First{/* тело класса*/}
```

3)

```
/*комментарий */
package first;
import java.util.*;
class First{/* тело класса*/}
```

Вопрос 6.2.

Какие определения интерфейса **MyInterface** являются корректными?

1)
```
interface MyInterface{
    public int result(int i){return(i++);}}
```
2)
```
interface MyInterface{
    int result(int i);}
```
3)
```
public interface MyInterface{
    public static int result(int i);}
```
4)
```
public interface MyInterface{
    class MyClass {}}
```
5)
```
public interface MyInterface{
    public final static int i;
    public abstract int result(int i);}
```

Вопрос 6.3.

Какие из объявлений корректны, если

```
class Owner{
    class Inner{
    } }
```

1) `new Owner.Inner();`
2) `Owner.new Inner();`

3) `new Owner.new Inner();`
4) `new Owner().new Inner();`
5) `Owner.Inner();`
6) `Owner().Inner().`

Вопрос 6.4.

Что будет выведено в результате компиляции и выполнения следующего кода?

```
abstract class Abstract {
      abstract Abstract meth();
}
class Owner {
      Abstract meth() {
            class Inner extends Abstract {
                  Abstract meth() {
                        System.out.print("inner ");
                        return new Inner();
                  }     }
            return new Inner();
      }
}
public abstract class Quest4 extends Abstract{
      public static void main(String a[]) {
            Owner ob = new Owner();
            Abstract abs = ob.meth();
            abs.meth();
      }
}
```

1) inner;
2) inner inner;
3) inner inner inner;
4) ошибка компиляции;
5) ошибка времени выполнения.

Вопрос 6.5.

```
class Quest5 {
  char A;          // 1
  void A() {}      // 2
  class A {}       // 3
}
```

В какой строке может возникнуть ошибка при компиляции данного кода?

1) 1;
2) 2;
3) 3;
4) компиляция без ошибок.