Глава 5

ВНУТРЕННИЕ КЛАССЫ

Внутри каждой большой задачи сидит маленькая, пытающаяся пробиться наружу.

Закон больших задач Хоара

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. Решение о включении одного класса внутрь другого может быть принято при слишком тесном и частом взаимодействии двух классов.

Одной из серьезных причин использования внутренних классов является возможность быть подклассом любого класса независимо от того, подклассом какого класса является внешний класс. Фактически при этом реализуется ограниченное множественное наследование со своими преимуществами и проблемами.

При необходимости доступа к защищенным полям и методам некоторого класса может появиться множество подклассов в разных пакетах системы. В качестве альтернативы можно сделать этот подкласс внутренним и методами класса-владельца предоставить доступ к интересующим защищенным полям и методам.

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, т. е. запустить *Двигатель* можно только посредством использования объекта *Корабль* из его 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 соответствующий 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). Доступ к элементам внутреннего класса возможен из внешнего 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:

/ # 1 # обьявление внутреннего класса и использование его в качестве поля как поля # Ship.java */*

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

Внутреннему классу совершенно не обязательно быть полем класса владельца. Внутренний класс может быть использован любым членом своего внешнего класса, а может и не использоваться вовсе.

/ # 2 # обьявление внутреннего класса # Ship.java */*

```
package by.bsu.inner;
public class Ship {
       // поля и конструкторы внешнего класса
       public class Engine { // определение внутреннего класса
              // поля и методы
              public void launch() {
                   System.out.println("Запуск двигателя");
              }
       } // конец обьявления внутреннего класса
       // методы внешнего класса
}
```

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.

При компиляции создается объектный модуль, соответствующий внутреннему классу, который 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. Доступ будет разрешен по имени в том числе и к полям, объявленным как **private**. 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:

/ # 3 # наследование от внешнего и внутреннего классов # RaceShip.java */*

```
public class RaceShip 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 был объявлен.

/ # 4 # наследование от внутреннего классf # Motor.java */*

```
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 на примере определения логически самостоятельной сущности **PhoneNumber**, дополняющей описание внешней, глобальной для нее сущности **Abonent**, приведен ниже. Внутренний класс **PhoneNumber** может оказаться достаточно сложным и обширным, поэтому простое включение его полей и методов в класс **Abonent** сделало бы последний весьма громоздким и трудным для восприятия.

/ # 4 # сокрытие реализации во внутреннем классе # Abonent.java */*

```
package by.bsu.inner;
public class Abonent {
          private long id;
          private String name;
          private String tariffPlan;
          private PhoneNumber phoneNumber; // ссылка на внутренний класс
          public Abonent (long id, String name) {
                    this.id = id;
                    this.name = name;
          }
          // объявление внутреннего класса
          private class PhoneNumber {
                    private int countryCode;
                    private int netCode;
                    private int number;
                    public void setCountryCode(int countryCode) {
                              // проверка на допустимые значения кода страны
                              this.countryCode = countryCode;
```

```
                }
                public void setNetCode(int netCode) {
                        // проверка на допустимые значения кода сети
                        this.netCode = netCode;
                }
                public int generateNumber() {
                        int temp = new java.util.Random().nextInt(10_000_000);
                        // проверка значения temp на совпадение в БД
                        number = temp;
                        return number;
                }
        } // окончание внутреннего класса
        public long getId() {
                return id;
        }
        public String getName() {
                return name;
        }
        public String getTariffPlan() {
                return tariffPlan;
        }
        public void setTariffPlan(String tariffPlan) {
                this.tariffPlan = tariffPlan;
        }
        public String getPhoneNumber() {
                if (phoneNumber != null) {
                        return ("+" + phoneNumber.countryCode + "-"
                                + phoneNumber.netCode + "-" + phoneNumber.number);
                } else {
                        return ("phone number is empty!");
                }
        }
        // соответствует шаблону Façade
        public void obtainPhoneNumber(int countryCode, int netCode) {
                phoneNumber = new PhoneNumber();
                phoneNumber.setCountryCode(countryCode);
                phoneNumber.setNetCode(netCode);
                phoneNumber.generateNumber();
        }
        @Override
        public String toString() {
                StringBuilder s = new StringBuilder(100);
                s.append("Abonent '" + name + "':\n");
                s.append("    ID - " + id + "\n");
                s.append("    Tariff Plan -  " + tariffPlan + "\n");
                s.append("    Phone Number - " + getPhoneNumber() + "\n");
                return s.toString();
        }
}
```

/ # 5 # инициализация и использование экземпляра Abonent # MobilMain.java */*

```
package by.bsu.inner.run;
import by.bsu.inner.Abonent;
public class MobilMain {
          public static void main(String[] args) {
                    Abonent abonent = new Abonent(819002, "Timofey Balashov");
                    abonent.setTariffPlan("free");
                    abonent.obtainPhoneNumber(375, 25);
                    System.out.println(abonent);
          }
}
```

В результате будет выведено:

Abonent 'Timofey Balashov**':**
ID - 819002
Tariff Plan - Free
Phone Number - +375–25–7492407

Внутренний класс определяет сущность предметной области «номер телефона» (класс **PhoneNumber**), которая обычно непосредственно связана в информационной системе с объектом класса **Abonent**. Класс **PhoneNumber** в данном случае определяет только способы доступа и изменения своих атрибутов и совершенно невидим вне класса **Abonent**, включающего метод по созданию и инициализации объекта внутреннего класса с составным номером телефона, способ построения которого скрыт от всех других классов.

Внутренний класс может быть объявлен также внутри метода или логического блока внешнего (owner) 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.

/ # 6 # внутренний класс, обьявленный внутри метода # AbstractTeacher.java # TeacherCreator.java # Teacher.java # TeacherLogic.java # Runner.java */*

```
package by.bsu.inner.study;
public abstract class AbstractTeacher {
         private int id;
         public AbstractTeacher(int id) {
                   this.id = id;
         }
         /* методы */
         public abstract boolean excludeStudent(String name);
}
```

```
package by.bsu.inner.study;
public class Teacher extends AbstractTeacher {
        public Teacher(int id) {
                super(id);
        }
        /* методы */
        @Override
        public boolean excludeStudent(String name) {
                 return false;
        }
}
package by.bsu.inner.study;
public class TeacherCreator {
        public static AbstractTeacher createTeacher(int id) {
           // объявление класса внутри метода
           class Rector extends AbstractTeacher {
                  Rector (int id) {
                    super(id);
                  }
                  @Override
                  public boolean excludeStudent(String name) {
                              if (name != null) { // изменение статуса студента в базе данных
                                    return true;
                              } else {
                                    return false;
                              }
                            }
           } // конец внутреннего класса
                  if (isRectorId(id)) {
                            return new Rector(id);
                  } else {
                            return new Teacher(id);
                  }
        }
        private static boolean isRectorId(int id) {
                  // проверка id
                  return id  == 6; // stub
        }
}
package by.bsu.inner.study;
public class TeacherLogic {
        public void excludeProcess(int rectorId, String nameStudent) {
                 AbstractTeacher teacher = TeacherCreator.createTeacher(rectorId);

                 System.out.println("Студент: " + nameStudent
                 + " отчислен:" + teacher.excludeStudent(nameStudent));
        }
}
package by.bsu.inner.study;
public class Runner {
```

```
        public static void main(String[ ] args) {
                TeacherLogic tl = new TeacherLogic();
                tl.excludeProcess(777, "Олейников");
                tl.excludeProcess(6, "Олейников");
        }
}
```

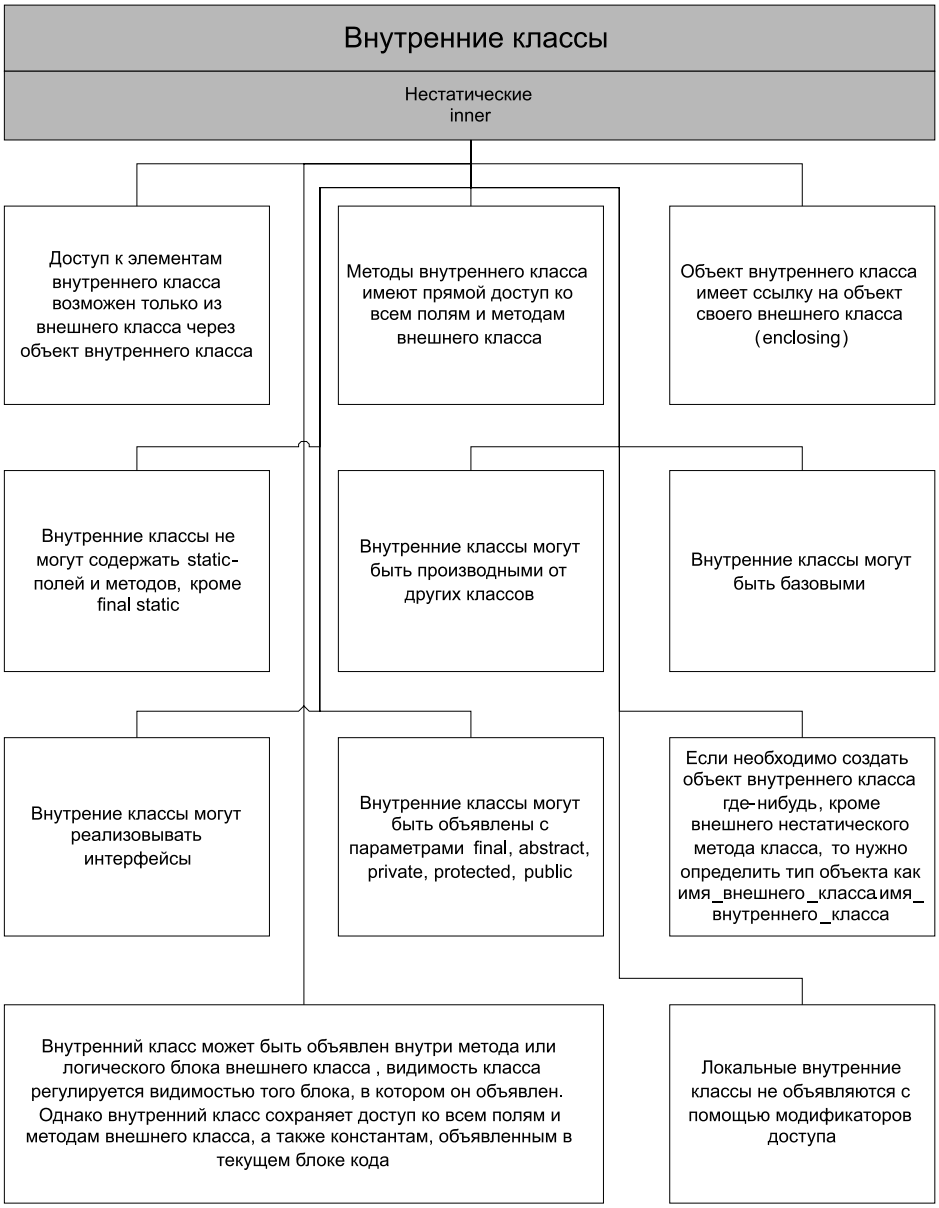

Рис. 5.1. *Внутренние классы*

В результате будет выведено:

Студент: Олейников отчислен: false
Студент: Олейников отчислен: true

Класс **Rector** объявлен в методе **createTeacher(int id)**, и, соответственно, объекты этого класса можно создавать только внутри 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, как в данном случае, при наследовании внутренним классом функциональности обычного класса, в частности, **AbstractTeacher**. При компиляции данного кода с внутренним классом ассоциируется объектный модуль со сложным именем **TeacherCreator$1Rector.class**, тем не менее однозначно определяющим связь между внешним и внутренним классами. Цифра **1** в имени говорит о том, что потенциально в других методах класса могут быть объявлены внутренние классы с таким же именем.

Вложенные (nested) классы

Если не существует жесткой 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. Если предполагается использовать внутренний класс в качестве подкласса, следует исключить использование в его теле любых прямых обращений к членам класса владельца.

/ # 7 # вложенный класс # Ship.java # RunnerShip.java */*

```
package by.bsu.nested;
public class Ship {
          private int id;
```

```
        // abstract, final, private, protected - допустимы
        public static class LifeBoat {
                private int boatId;
                public static void down() {
                        System.out.println("шлюпки на воду!");
                }
                public void swim() {
                        System.out.println("отплытие шлюпки");
                }
        }
}
package by.bsu.nested;
public class RunnerShip {
        public static void main(String[ ] args) {
                // вызов статического метода
                Ship.LifeBoat.down();
                // создание объекта статического класса
                Ship.LifeBoat lifeBoat = new Ship.LifeBoat();
                // вызов обычного метода
                lifeBoat.swim();
        }
}
```

Статический метод вложенного класса вызывается при указании полного относительного пути к нему. Объект **lifeBoat** 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.

/ # 8 # класс, вложенный в интерфейс # LearningDepartment.java # University.java */*

```
package by.bsu.nested;
public interface University {
        int NUMBER_FACULTY = 20;
        void create();
        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.

Рис. 5.2. *Вложенные классы*

Анонимные (anonymous) классы

Анонимные (безымянные) классы применяются для придания уникальной функциональности отдельно взятому экземпляру, для обработки событий, реализации блоков прослушивания, реализации интерфейсов, запуска потоков 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.

Конструктор анонимного класса определить невозможно.

/ # 9 # анонимные классы # WrapperString.java # Runner.java */*

```
package by.bsu.anonym;
public class WrapperString {
          private String str;
          public WrapperString(String str) {
                    this.str = str;
          }
          public String getStr() {
                    return str;
          }
          public String replace(char oldChar, char newChar) { // замена первого символа
                    char[] array = new char[str.length()];
                    str.getChars(0, str.length(), array, 0);
                    for (int i = 0; i < array.length; i++) {
                              if (array[i] == oldChar) {
                                        array[i] = newChar;
                                        break;
                              }
                    }
                    return new String(array);
          }
}
package by.bsu.anonym;
public class Runner {
          public static void main(String[ ] args) {
                     String ob = "qweRtRRR";
                     WrapperString wrFirst = new WrapperString(ob);
                     // анонимный класс #1
                     WrapperString wrLast = new WrapperString(ob) {
                               // замена последнего символа
                               @Override
                               public String replace(char oldChar, char newChar) {
                                        char[] array = new char[getStr().length()];
                                        getStr().getChars(0, getStr().length(), array, 0);
                                        for (int i = array.length - 1; i > 0; i--) {
                                                  if (array[i] == oldChar) {
                                                            array[i] = newChar;
                                                            break;
                                                  }
                                        }
                                        return new String(array);
                               }
                     }; // конец объявления анонимного класса
                     WrapperString wr2 = new WrapperString(ob) { // анонимный класс #2
                               private int position = 2; // собственное поле
                               // замена символа по позиции
                               public String replace(char oldChar, char newChar) {
                                        int counter = 0;
```

```
                                        char[] array = new char[getStr().length()];
                                        getStr().getChars(0, getStr().length(), array, 0);
                                        if (verify(oldChar, array)) {
                                                for (int i = 0; i < array.length; i++) {
                                                        if (array[i] == oldChar) {
                                                                counter++;
                                                                if (counter == position) {
                                                                        array[i] = newChar;
                                                                                break;
                                                                }
                                                        }
                                                }
                                        }
                                        return new String(array);
                                }
                                // собственный метод
                                public boolean verify(char oldChar, char[] array){
                                        int counter = 0;
                                        for (char c : array) {
                                                if (c == oldChar) {
                                                    counter++;
                                                }
                                        }
                                        return counter >= position;
                                }
                        }; // конец объявления анонимного класса
                        System.out.println(wrLast.replace('R', 'Y'));
                        System.out.println(wr2.replace('R', 'Y'));
                        System.out.println(wrFirst.replace('R', 'Y'));
                }
}
```

В результате будет выведено:

qweRtRRY
qweRtYRR
qweYtRRR

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

/ # 10 # анонимный класс в перечислении # Shape.java # EnumRunner.java */*

```
package by.bsu.enums;
public enum Shape {
        RECTANGLE, SQUARE, TRIANGLE { // анонимный класс
                public double computeSquare() { // версия для TRIANGLE
                        return this.getA() * this.getB() / 2;
                }
        };
        private double a;
        private double b;
        public double getA() {
                return a;
        }
        public double getB() {
                return b;
        }
        public void setShape(double a, double b) {
                if ((a <= 0 || b <= 0) || a != b && this == SQUARE) {
                        throw new IllegalArgumentException();
                }
                this.a = a;
                this.b = b;
        }
        public double computeSquare() { // версия для RECTANGLE и SQUARE
                return a * b;
        }
        public String toString() {
                return name() + "-> 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 + " площадь= " + f.computeSquare());
                }
        }
}
```

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

Рис. 5.3. *Анонимные классы*

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

Ситуации, в которых следует использовать внутренние классы:

- выделение самостоятельной логической части сложного класса;
- сокрытие реализации;
- одномоментное использование переопределенных методов. В том числе обработка событий;
- запуск потоков выполнения;
- отслеживание внутреннего состояния, например, с помощью **enum**.

Анонимные классы, как и остальные внутренние, допускают вложенность друг в друга, что может сильно запутать код и сделать эти конструкции непонятными, поэтому в практике программирования данная техника не используется.

Задания к главе 5

Вариант А

1. Создать класс **Notepad** с внутренним классом или классами, с помощью объектов которого могут храниться несколько записей на одну дату.

2. Создать класс **Payment** с внутренним классом, с помощью объектов которого можно сформировать покупку из нескольких товаров.
3. Создать класс **Account** 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** 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** с внутренним классом, с помощью объектов которого можно хранить информацию об истории выдач книги читателям.
7. Создать класс **Европа** с внутренним классом, с помощью объектов которого можно хранить информацию об истории изменения территориального деления на государства.
8. Создать класс **City** с внутренним классом, с помощью объектов которого можно хранить информацию о проспектах, улицах, площадях.
9. Создать класс **BlueRayDisc** 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** с внутренним классом, с помощью объектов которого можно хранить информацию об отделах, товарах и услугах.
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** с внутренним классом, с помощью объектов которого можно хранить информацию об операционной системе, процессоре и оперативной памяти.
16. Создать класс **Park** с внутренним классом, с помощью объектов которого можно хранить информацию об аттракционах, времени их работы и стоимости.
17. Создать класс **Cinema** с внутренним классом, с помощью объектов которого можно хранить информацию об адресах кинотеатров, фильмах и времени начала 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.