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, применение внутренних классов есть один из способов *сокрытия кода*, так как внутренний класс может быть недоступен и не виден вне класса-владельца. Внутренние классы также могут использоваться в качестве *блоков прослушивания событий*. Решение о включении одного класса внутрь другого может быть принято при тесном и частом взаимодействии двух классов.

Одной из 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 *Студент*, *Адрес*. Объект класса *Адрес* как единое целое характеризует объект класса *Студент,* расположен внутри (невидим извне) объекта *Студент*. Его состояние есть часть состояния *Студента*. Оба этих объекта связаны. Перед инициализацией объекта внутреннего класса *Адрес* должен быть создан объект внешнего класса *Студент*. Классы связаны описанием общего состояния.

Если необходимо определить и связать класс с некоторой функциональностью, очень близкой этому классу, то применяется статическое вложение класса, делающее независимым объект с вложенной функциональностью от

класса-владельца, но логически через имя внешенего класса связывает с ним. Объект такого класса можно создать, не создавая объект класса-владельца.

Такие статические вложенные классы объявляются с модификатором **static**. Статические классы могут обращаться к нестатическим членам включающего класса не напрямую, а только через его объект.

Нестатические внутренние классы имеют доступ ко всем переменным и методам своего внешнего класса-владельца и требуют последовательного создания объектов внешнего и внутреннего классов.

Применение анонимных классов и их подмножества: лямбда-выражений, позволяет сократить количество кода. Анономные классы сокращают число подклассов, лямбда-выражения записываются еще короче и с более высоким акцентом на функционал объекта.

Внутренние (inner) классы

Нестатические вложенные классы принято называть внутренними, или *inner* классами.

Связь между внешним и внутренним классами при этом определяется необходимостью привязывания логической сущности внутреннего класса к сущности внешнего класса.

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, или enclosing объектом.

Пусть изначально описание класса **Student** представлено в виде:

```
package by.epam.learn.study;
public class Student {
    private int studentId;
    private String name;
    private int group;
    private String faculty;
    private String city;
    private String street;
    private int houseId;
    private int flatId;
    private String email;
    private String skype;
    private long phoneNumber;
    // constructors, methods
}
```

В описании класса можно целый набор данных объединить под общим именем **Address** и выделить его как внутренний класс. Класс **Student** станет более коротким и понятным.

Внешний и внутренний классы могут выглядеть в итоге так:

/ # 1 # объявление внутреннего класса и использование его в качестве поля # Student.java */*

```
package by.epam.learn.study;
public class Student {
    private int studentId;
    private String name;
    private int group;
    private String faculty;
    private Address address;
    // private, protected – may be
    public class Address { // inner class: begin
        private String city;
        private String street;
        private int houseId;
        private int flatId;
        private String email;
        private String skype;
        private long phoneNumber;
        // more code
    }// inner class: end
}
```

Внутренний класс может быть использован любым членом своего внешнего класса, а может и не использоваться вовсе, хотя в этом случае утрачивается его смысл.

Использование объекта внутреннего класса вне своего внешнего класса возможно только при наличии доступа (видимости) и при объявлении ссылки в виде:

```
Student.Address address = new Student().new Address();
```

Здесь сначала создается объект внешнего класса, а затем объект внутреннего класса. Другим способом объект внутреннего класса создать не получится. Объекту внутреннего класса совершенно не обязательно быть полем класса-владельца. Основное отличие от внешнего класса состоит в больших возможностях ограничения видимости и сокрытия реализации 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 **address**, 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 внутренний класс получает собственный модуль для интерпретации, соответствующий внутреннему классу, который получит имя

Student$Address.class. Внешний же класс будет скомпилирован в обычный файл **Student.class**.

Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса, как будто они его собственные, 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 поля и методы, кроме **final static**. Внутренние классы имеют право наследовать другие классы, реализовывать интерфейсы и выступать в роли объектов наследования. Допустимо наследование следующего вида:

/ # 2 # наследование от внешнего и внутреннего классов # SubStudent.java */*

```
package by.epam.learn.study;
public class SubStudent extends Student {
   // code
   public class SubAddress extends Address {
     // code
  }
}
```

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.

/ # 3 # наследование от внутреннего класса # FreeAddress.java */*

```
package by.epam.learn.study;
public class FreeAddress extends Student.Address {
  public FreeAddress() {
    new Student().super();
  }
  public FreeAddress(Student student) {
    student.super();
  }
}
```

В данном случае конструктор класса **FreeAddress** должен объявлять объект класса **Student** или получать его в виде параметра, что позволит получить доступ к ссылке на внутренний класс **Address**, наследуемый классом **FreeAddress**.

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

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

```
class Outer {
   private class Inner {}
}
```

Поля внешнего класса видны внутреннему классу так, будто они его собственные, модификаторы видимости игнорируются. Применить ссылку **this** также не получится, так как **this** внутри класса **Inner** указывает на его собственный объект, и ни в коем случае не на его владельца. Поэтому **this.id** будет давать ошибку компиляции.

/ # 4 # доступ к полям внешнего класса # Owner.java */*

```
public class Owner {
  private int id;
  public class Inner {
    public void buildId() {
      id += 1000;
    }
  }
}
```

Внешний класс напрямую не видит никаких компонентов своего внутреннего класса.

Вопрос доступа к полям внешнего класса можно рассмотреть путем объявления полей с одинаковыми именами во внутреннем и внешнем классах, чего на практике делать не рекомендуется.

/ # 5 # некорректное объявление полей классов # DumberOwner.java */*

```
public class DumberOwner {
  private int id;
  public class DumberInner {
    private int id;
    public void buildId(int id) {
      this.id = id + 100 * DumberOwner.this.id;
    }
  }
}
```

Поле **id** объявлено как поле класса **DumberInner** и при попытке доступа к полю **id** класса **DumberOwner** обращения типа **id** или **this.id** приведут к обращению к полю внутреннего класса. Для доступа к **id** внешнего класса нужно указать на имя класса, которому принадлежит объект, а именно **DumberOwner.this.id.**

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, объявленный внутри метода # AbstractTeacher.java
 # TeacherCreator.java # Teacher.java # TeacherLogic.java # StudyMain.java */
```

```
package by.epam.learn.study;
public abstract class AbstractTeacher {
  private int id;
  public AbstractTeacher(int id) {
    this.id = id;
  }
  public abstract boolean remandStudent(Student student);
}
package by.epam.learn.study;
public class Teacher extends AbstractTeacher {
  public Teacher(int id) {
    super(id);
  }
  @Override
  public boolean remandStudent(Student student) {
    return false;
  }
}
package by.epam.learn.study;
public class TeacherCreator {
  public static AbstractTeacher createTeacher(int id) {
    int value = 0;
    // class declaration inside a method
    class Rector extends AbstractTeacher {
      Rector(int id) {
        super(id);
      }
      @Override
      public boolean remandStudent(Student student) {
        // value++; compile error
        boolean result = false;
        if (student != null) {
          // student status change code in the database
          result =  true;
        }
        return result;
      }
    } // inner class: end
    if (isRectorId(id)) {
      return new Rector(id);
    } else {
      return new Teacher(id);
    }
  }
  private static boolean isRectorId(int id) {
    // checking id in the database
```

```
    return (id == 6); // stub
  }
}
package by.epam.learn.study;
public class TeacherLogic {
  public void expelledProcess(int rectorId, Student student) {
    AbstractTeacher teacher = TeacherCreator.createTeacher(rectorId);
    boolean result = teacher.remandStudent(student);
    System.out.println("Student expelled: " + result);
  }
}
package by.epam.learn.study;
public class StudyMain {
  public static void main(String[] args) {
    TeacherLogic logic = new TeacherLogic();
    Student student = new Student();
    logic.expelledProcess(42, student);
    logic.expelledProcess(6, student);
  }
}
```

В результате будет выведено:

Student expelled: false
Student expelled: true

Класс **Rector** объявлен в методе **createTeacher(int id)** и, соответственно, объекты этого класса можно создавать только внутри метода, из любого другого метода или конструктора внешнего класса внутренний класс недоступен. Однако существует единственная 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 в других методах класса также можно объявить внутренний класс с таким же именем.

Свойства внутренних классов:

- Доступ к элементам внутреннего класса возможен только из внешнего класса через объект внутреннего класса;
- Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса;
- Объект внутреннего класса имеет ссылку на объект своего внешнего класса (enclosing);
- Внутренние классы не могут содержать **static**-полей и методов, кроме **final static**;
- Внутренние классы могут быть производными от других классов;

- Внутренние классы могут быть суперклассами;
- Внутренние классы могут реализовывать интерфейсы;
- Внутренние классы могут быть объявлены с параметрами **final**, **abstract**, **private**, **protected**, **public**;
- Если необходимо создать объект внутреннего класса где-нибудь, кроме внешнего нестатического метода класса, то нужно определить тип объекта как *OwnerType.InnerType*;
- Внутренний класс может быть объявлен внутри метода или логического блока внешнего класса, видимость класса регулируется видимостью того блока, в котором он объявлен. Однако внутренний класс сохраняет доступ ко всем полям и методам внешнего класса, а также **final**-переменным, объявленным в текущем блоке кода;
- Локальному внутреннему классу, объявленному внутри метода или логического блока, модификатор доступа не требуется, так как он все равно не доступен напрямую вне метода.

Вложенные (nested) классы

Если не существует жесткой необходимости в одновременном обязательном существовании объекта внутреннего класса и объекта внешнего класса, то есть смысл сделать такой внутренний класс статическим, который будет тогда называться вложенным, или *nested* классом.

Связь между внешним и внутренним классами определяется необходимостью привязывания логической функциональности внутреннего класса.

Вложенный класс логически связан с классом-владельцем, но его объект может быть использован независимо от объекта внешнего класса. Такой класс обычно определяет дополнительный функционал для класса-владельца.

При объявлении такого внутреннего класса присутствует служебное слово **static**, и такой класс называется вложенным (*nested*). Если класс объявлен внутри интерфейса, то он получает спецификаторы **public static** 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 иметь 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-компаратор # Student.java # ComparingMain.java */*

```
package by.epam.learn.nested;
import java.util.Comparator;
public class Student {
  private int studentId;
  private String name;
  private int group;
  private float averageMark;
  public Student(int studentId, String name, int group, float averageMark) {
    this.studentId = studentId;
    this.name = name;
    this.group = group;
    this.averageMark = averageMark;
  }
  public String getName() {
    return name;
  }
  public int getGroup() {
    return group;
  }
  public float getAverageMark() {
    return averageMark;
  }
  // nested classes
  public static class GroupComparator implements Comparator<Student> {
    @Override
    public int compare(Student o1, Student o2) {
      return o1.group - o2.group;
    }
  }
  public static class NameComparator implements Comparator<Student> {
    @Override
    public int compare(Student o1, Student o2) {
      return o1.name.compareTo(o2.name);
    }
  }
  public static class MarkComparator implements Comparator<Student> {
    @Override
    public int compare(Student o1, Student o2) {
      return Float.compare(o2.averageMark, o1.averageMark);
    }
  }
}
package by.epam.learn.nested;
import java.util.Comparator;
public class ComparingMain {
  public static void main(String[] args) {
    Student st1 = new Student(2341757, "Mazaliyk", 3, 5.42f);
```

```
    Student st2 = new Student(2341742, "Polovinkin", 1, 5.42f);
    // creating a static class object
    Student.NameComparator nameComparator = new Student.NameComparator();
    int result1 = nameComparator.compare(st1, st2);
    System.out.println(st1.getName() + " [" + result1 + "] " + st2.getName());
    Student.MarkComparator markComparator = new Student.MarkComparator();
    int result2 = markComparator.compare(st1, st2);
    System.out.println(st1.getAverageMark() +" [" +  result2+ "] "
                                              + st2.getAverageMark());
    Student.GroupComparator groupComparator = new Student.GroupComparator();
    int result3 = groupComparator.compare(st1, st2);
    System.out.println(st1.getGroup() + " [" + result3+ "] " + st2.getGroup());
  }
}
```

Результатом будет:

Mazaliyk [-3] Polovinkin
5.42 [0] 5.42
3 [2] 1

Объект **nameComparator** вложенного класса создается с использованием имени внешнего класса без вызова его конструктора. Если во вложенном классе объявлен статический метод, то он просто вызывается при указании полного относительного пути к нему.

/ # 8 # видимость полей # Owner.java */*

```
public class Owner {
  private int value = 1;
  static int statValue = 2;
   public static class Nested  {
     {
       statValue++;
       // value++; invisible
     }
  }
}
```

Статическому классу доступно только статическое содержимое класса-владельца.

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.

/ # 9 # класс, вложенный в интерфейс # Logic.java # NestedLogic.java */*

```
package by.epam.learn.nested;
public interface Logic {
  void doLogic();
```

```
  class NestedLogic { // public static: default parameters
    public long value;
    public NestedLogic() { /* code */ }
    public static void assign() { /* code */ }
    public void accept() { /* code */ }
  }
}
```

Такой внутренний класс использует пространство имен интерфейса. Вызов статического метода выглядит так:

```
Logic.NestedLogic.assign();
```

Свойства вложенных классов:

- Вложенный класс может быть базовым, производным, реализующим интерфейсы;
- Статический вложенный класс для доступа к нестатическим членам и методам внешнего класса должен создавать объект внешнего класса;
- Вложенный класс имеет доступ к статическим полям и методам внешнего класса;
- Подкласс вложенного класса не наследует возможность доступа к членам внешнего класса, которыми наделен его суперкласс;
- Класс, вложенный в интерфейс, статический по умолчанию;
- Статический метод вложенного класса вызывается при указании полного относительного пути к нему.

Анонимные (anonymous) классы

Анонимные (безымянные) внутренние 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**.

С появлением функциональных интерфейсов появилось понятие анонимного объекта-функции, то есть объекта, основное назначение которого передать реализацию конкретной функциональности в анонимном виде:

```
FunctionalInterface lambda = () -> doAction();
```

Функциональные интефейсы рассматривались в предыдущей главе.

Анонимные классы эффективно используются, как правило, для реализации (переопределения) одного или нескольких методов. Этот прием эффективен

в случае, когда необходимо переопределение метода, но создавать новый класс нет необходимости из-за узкой области или одномоментного применения объекта.

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.

Конструктор анонимного класса определить невозможно. Нельзя создать анонимный класс для **final**-класса.

/ # 10 # анонимные классы # StudentAction.java # StudentActionMain.java */*

```
package by.epam.learn.inner.anonymous;
public class StudentAction {
  private final static int BASE_COEFFICIENT = 6;
  public double defineScholarship(float averageMark) {
    double value = 100;
    if (averageMark > BASE_COEFFICIENT) {
      value *= 1 + (BASE_COEFFICIENT / 10.0);
    }
    return value;
  }
}
package by.epam.learn.inner.anonymous;
public class StudentActionMain {
  public static void main(String[] args) {
    StudentAction action = new StudentAction();// usually object
    StudentAction actionAnon = new StudentAction() {// anonymous class object
      int base = 9; // invisible
      @Override
      public double defineScholarship(float averageMark) {
        double value = 100;
        if (averageMark > base) {
          value *= 1 + (base / 10.0);
        }
        return value;
      }
    };
    System.out.println(action.defineScholarship(9.05f));
    System.out.println(actionAnon.defineScholarship(9.05f));
  }
}
```

В результате будет выведено:

```
160.0
190.0
```

Анонимный класс может использовать только неизменяемые параметры и локальные переменные метода, в котором он создан. При запуске приложения происходит объявление объекта **actionAnon** с применением анонимного класса,

в котором переопределяется метод **defineScholarship()**. Вызов данного метода на объекте **actionAnon** приводит к вызову версии метода из анонимного класса, который компилируется в объектный модуль с именем **StudentActionMain$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 анонимном классе разрешено объявлять собственные поля и методы, которые не 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.

/ # 11 # анонимный класс в перечислении # Shape.java # EnumMain.java */*

```
package by.epam.learn.inner.anonymous;
import java.util.StringJoiner;
public enum Shape {
  RECTANGLE (2, 3) {
    public double computeSquare() {
      return this.getA() * this.getB();
    }
  }, TRIANGLE (2, 3) {
    public double computeSquare() {
      return this.getA() * this.getB() / 2;
    }
  };
  private double a;
  private double b;

  Shape(double a, double b) {
    this.a = a;
    this.b = b;
  }
  public double getA() {
    return a;
  }
  public void setA(double a) {
    this.a = a;
  }
  public double getB() {
    return b;
  }
  public void setB(double b) {
    this.b = b;
  }
  public abstract double computeSquare();
  @Override
```

```
  public String toString() {
    return new StringJoiner(", ", Shape.class.getSimpleName() + "[", "]")
            .add("a=" + a).add("b=" + b).toString();
  }
}
package by.epam.learn.inner.anonymous;
import java.util.Arrays;
public class EnumMain {
  public static void main(String[] args) {
    Arrays.stream(Shape.values()).forEach(s -> System.out.println(s.computeSquare()));
  }
}
```

В результате будет выведено:

6.0
3.0

Свойства анонимных классов:

- расширяет другой класс или реализует интерфейс при объявлении одного единственного объекта, остальным объектам будет соответствовать реализация, определенная в самом классе;
- объявление анонимного объекта выполняется одновременно с созданием его объекта с помощью оператора **new**;
- конструкторы анонимных классов ни определить, ни переопределить нельзя;
- анонимные классы допускают вложенность друг в друга (нежелательно использовать);
- объявление анонимного класса в перечислении отличается от простого анонимного класса, поскольку инициализация всех элементов происходит при первом обращении к типу.

Ситуации, в которых следует использовать внутренние классы:

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

Вопросы к главе 5

1. Что такое внутренние, вложенные и анонимные классы? Как определить классы такого вида? Как создать объекты классов такого вида.
2. Перечислить возможности доступа к членам внешнего класса, которым наделены вложенные классы?

3. Перечислить возможности доступа к членам внешнего класса, которым наделены внутренние классы?
4. Перечислить возможности доступа к членам внешнего класса, которым наделены анонимные классы?
5. Могут ли классы внутри классов быть базовыми, производными или реализующими интерфейсы?
6. Как решить проблему множественного наследования с применением внутренних классов?
7. Можно ли анонимный класс создать от абстрактного класса?
8. Можно ли анонимный класс создать от **final**-класса?
9. Во что компилируется анонимный внутренний класс в классе? В методе?
10. Можно ли создать анонимный статический внутренний класс?
11. Как получить доступ к внутреннему классу, объявленному внутри метода извне метода?

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.

Тестовые задания к главе 5

Вопрос 5.1.

Дано:

```
class Garden {
  public static class Plant {}
}
```

Выбрать корректное объявление объекта внутреннего класса (выбрать один).

a) Plant plant = new Plant();
b) Garden.Plant plant = new Garden.Plant();
c) Plant plant = new Garden.Plant();
d) Garden.Plant plant = new Garden().new Plant();

Вопрос 5.2.

Дан код:

```
class Outer {
  public int size = 0;
  static class Inner {
    public void incrementSize() {
      /* line 1*/ += 1;
    }
  }
}
```

Что необходимо записать вместо комментария *line 1*, чтобы код класса Inner компилировался без ошибок? (выбрать один)

a) size
b) Outer.size
c) new Outer().size
d) Outer.Inner.size
e) нет верного варианта

Вопрос 5.3.

Дано:

```
class Outer{
  class Inner{}
  public void outerMethod() {
    // место создания объекта класса Inner
  }
}
```

Какой вариант вызовет ошибку компиляции при создании объекта класса Inner? (выбрать один)

a) Inner a = new Inner();
b) Inner b = Outer.new Inner();
c) Inner c = new Outer().new Inner();
d) Outer.Inner d = new Outer().new Inner();
e) Outer.Inner e = new Inner();

Вопрос 5.4.

Дан код:

```
class Clazz{
  {System.out.print("clazz ");}
  public void method() {
    System.out.print("method1 ");
  }
}
```

```
class Owner{
  {System.out.print("owner ");}
  private int N=10;
  class Inner extends Clazz {
    {System.out.print("inner ");}
    public void method(){
      System.out.print("method2 ");
    }
  }
}
class Quest {
  public static void main(String[] args) {
    new Owner().new Inner().method();
  }
}
```

Каким будет результат компиляции и запуска приложения? (выбрать один)

a) compilation fails
b) clazz owner inner method2
c) owner clazz inner method2
d) owner clazz inner method1

Вопрос 5.5.

Дан код:

```
class Clazz {
  static int i = 1;
}
class Outer {
  class Inner extends Clazz {
  }
}
class Quest {
  public static void main(String[] args) {
    System.out.println(_________);
  }
}
```

Какое обращение к переменной *i* корректно из метода *main*? (выбрать один)

a) Outer.Inner.Clazz.i
b) new Outer.Inner().i
c) Outer.Inner.super.i
d) Outer.Inner.i