Глава 3
КЛАССЫ

Класс представляет описание совокупности объектов с общими атрибутами, методами, отношениями и семантикой.

Классы – основной элемент абстракции языка Java, основное назначение которого, кроме реализации назначенного ему контракта, это сокрытие реализации. Классы всегда взаимодействуют друг с другом и объединяются в пакеты. Из пакетов создаются модули, которые взаимодействуют друг с другом только через ограниченное количество методов и классов, не имея никакого представления о процессах, происходящих внутри других модулей.

Имя класса в пакете должно быть уникальным. Физически пакет представляет собой каталог, в который помещаются программные файлы, содержащие реализацию классов.

Классы позволяют провести декомпозицию поведения сложной системы до множества элементарных взаимодействий связанных объектов. Класс определяет структуру и/или поведение некоторого элемента предметной области, для которой разрабатывается программная модель.

Определение простейшего класса без наследования имеет вид:

```
class ИмяКласса {
    {}//логические блоки
    // дружественные данные и методы
    private// закрытые данные и методы
    protected// защищенные данные и методы
    public// открытые данные и методы
}
```

Переменные класса и константы

Классы инкапсулируют переменные и методы – члены класса. Переменные класса объявляются в нем следующим образом:

```
спецификатор тип имя;
```

В языке Java могут использоваться статические переменные класса, объявленные один раз для всего класса со спецификатором **static** и одинаковые для всех экземпляров (объектов) класса, или переменные экземпляра класса, создаваемые для каждого объекта класса. Поля класса объявляются со спецификаторами доступа **public**, **private**, **protected** или по умолчанию без спецификатора. Кроме данных – членов класса, в методах класса используются локальные переменные и параметры методов. В отличие от переменных класса, инкапсулируемых нулевыми элементами, переменные методов не инициализируются по умолчанию.

Переменные со спецификатором **final** являются константами. Спецификатор **final** можно использовать для переменной, объявленной в методе, а также для параметра метода.

В следующем примере рассматриваются объявление и инициализация значений полей класса и локальных переменных метода, а также использование параметров метода:

```
/* пример # 1 : типы атрибутов и переменных: Second.java */
package chapt03;
import java.util.*;

class Second {
     private int x; // переменная экземпляра класса
     private int y = 71; // переменная экземпляра класса
     public final int CURRENT_YEAR = 2007; // константа
     protected static int bonus; // переменная класса
     static String version = "Java SE 6"; // переменная класса
     protected Calendar now;

     public int method(int z) { // параметр метода
          z++;
          int a; // локальная переменная метода
          //a++; // ошибка компиляции, значение не задано
          a = 4; //инициализация
          a++;
          now = Calendar.getInstance();//инициализация
          return a + x + y + z;
     }
}
```

В рассмотренном примере в качестве переменных экземпляра класса, переменных класса и локальных переменных метода использованы данные базовых типов, не являющиеся ссылками на объекты (кроме **String**). Данные могут быть ссылками, назначить которым реальные объекты можно с помощью оператора **new**.

Ограничение доступа

Язык Java предоставляет несколько уровней защиты, обеспечивающих возможность настройки области видимости данных и методов. Из-за наличия пакетов Java работает с четырьмя категориями видимости между элементами классов:

- по умолчанию – дружественные члены класса доступны классам, находящимся в том же пакете;
- **private** – члены класса доступны только членам данного класса;
- **protected** – члены класса доступны классам, находящимся в том же пакете, и подклассам – в других пакетах;
- **public** – члены класса доступны для всех классов в этом и других пакетах.

Член класса (поле или метод), объявленный **public**, доступен из любого места вне класса. Все, что объявлено **private**, доступно только методам внутри класса и нигде больше. Если у элемента вообще не указан модификатор уровня доступа, то такой элемент будет виден и доступен из подклассов и классов того же пакета. Именно такой уровень доступа используется по умолчанию. Если же необходимо, чтобы элемент был доступен из другого пакета, но только подклассам того класса, которому он принадлежит, нужно объявить такой элемент со спецификатором **protected**. Действие спецификаторов доступа распространяется только на тот элемент класса, перед которым они стоят.

Спецификатор доступа **public** может также стоять перед определением внешнего (enclosing) класса. Если данный спецификатор отсутствует, то класс недоступен из других пакетов.

Конструкторы

Конструктор – это метод, который автоматически вызывается при создании объекта класса и выполняет действия по инициализации объекта. Конструктор имеет то же имя, что и класс; вызывается не по имени, а только вместе с ключевым словом **new** при создании экземпляра класса. Конструктор не возвращает значение, но может иметь параметры и быть перегружаемым.

Деструкторы в языке Java не используются, объекты уничтожаются сборщиком мусора после прекращения их использования (потери ссылки). Аналогом деструктора является метод **finalize()**. Исполняющая среда языка Java будет вызывать его каждый раз, когда сборщик мусора будет уничтожать объект класса, которому не соответствует ни одна ссылка.

```
/* пример # 2 : перегрузка конструктора: Quest.java */
package chapt03;

public class Quest {
      private int id;
      private String text;
      // конструктор без параметров (по умолчанию)
      public Quest() {
          super();/* если класс будет объявлен без конструктора, то
                     компилятор предоставит его именно в таком виде*/
      }
      // конструктор с параметрами
      public Quest(int idc, String txt) {
            super();/*вызов конструктора суперкласса явным образом
            необязателен, компилятор вставит его автоматически*/
            id = idc;
            text = txt;
      }
}
```

Объект класса **Quest** может быть создан двумя способами, вызывающими один из конструкторов:

```
Quest a = new Quest();//инициализация полей значениями по умолчанию
Quest b = new Quest(71, "Сколько бит занимает boolean?");
```

Оператор **new** вызывает конструктор, поэтому в круглых скобках могут стоять аргументы, передаваемые конструктору.

Если конструктор в классе не определен, Java предоставляет конструктор по умолчанию без параметров, который инициализирует поля класса значениями по умолчанию, например: **0**, **false**, **null**. Если же конструктор с параметрами определен, то конструктор по умолчанию становится недоступным и для его вызова необходимо явное объявление такого конструктора. Конструктор подкласса всегда вызывает конструктор суперкласса. Этот вызов может быть явным или неявным и всегда располагается в первой строке кода конструктора. Если конструктору суперкласса нужно передать параметры, то необходим явный вызов:

```
super(параметры);
```

В следующем примере объявлен класс **Point** с двумя полями (атрибутами), конструктором и методами для инициализации и извлечения значений атрибутов.

```
/* пример # 3 : вычисление расстояния между точками: Point.java:
LocateLogic.java: Runner.java */
package chapt03;

public class Point {
/* объект инициализируется при создании и не изменяется */
      private final double x;
      private final double y;

      public Point(final double xx, final double yy) {
            super();
            x = xx;
            y = yy;
      }
      public double getX() {
            return x;
      }
      public double getY() {
            return y;
      }
}
package chapt03;

public class LocateLogic {
      public double calculateDistance(
                                    Point t1, Point t2) {
            /* вычисление расстояния */
            double dx = t1.getX() - t2.getX();
            double dy = t1.getY() - t2.getY();
            return Math.hypot(dx, dy);
      }
}
package chapt03;
```

```
public class Runner {
      public static void main(String[] args) {
      // локальные переменные не являются членами класса
            Point t1 = new Point(5, 10);
            Point t2 = new Point(2, 6);
            System.out.print("расстояние равно : "
            + new LocateLogic().calculateDistance(t1, t2));
      }
}
```

В результате будет выведено:

```
расстояние равно : 5.0
```

Конструктор объявляется со спецификатором **public**, чтобы была возможность вызывать его при создании объекта в любом пакете приложения. Спецификатор **private** не позволяет создавать объекты вне класса, а спецификатор «по умолчанию» – вне пакета. Спецификатор **protected** позволяет создавать объекты в текущем пакете и для подклассов в других пакетах.

Методы

Изобретение методов является вторым по важности открытием после создания компьютера. Метод – основной элемент структурирования хода.

Все функции Java объявляются только внутри классов и называются методами. Простейшее определение метода имеет вид:

```
returnType methodName(список_параметров) {
    // тело метода
    return value;  // если нужен возврат значения (returnType не void)
}
```

Если метод не возвращает значение, ключевое слово **return** может отсутствовать, тип возвращаемого значения в этом случае будет **void**. Вместо пустого списка параметров метода тип **void** не указывается, а только пустые скобки. Вызов методов осуществляется из объекта или класса (для статических методов):

```
objectName.methodName();
```

Методы-конструкторы по имени вызываются автоматически только при создании объекта класса с помощью оператора **new**.

Для того чтобы создать метод, нужно внутри объявления класса написать объявление метода и затем реализовать его тело. Объявление метода как минимум должно содержать тип возвращаемого значения (возможен **void**) и имя метода. В приведенном ниже объявлении метода элементы, заключенные в квадратные скобки, являются необязательными.

```
[доступ] [static] [abstract] [final] [native]
[synchronized] returnType methodName(список_параметров)
                                [throws список_исключений]
```

Как и для полей класса, спецификатор доступа к методам может быть **public**, **private**, **protected** и по умолчанию. При этом методы суперкласса можно перегружать или переопределять в порожденном подклассе.

Объявленные в методе переменные являются локальными переменными метода, а не членами классов, и не инициализируются значениями по умолчанию при создании объекта класса или вызове метода.

Статические методы и поля

Поля данных, объявленные в классе как **static**, являются общими для всех объектов класса и называются переменными класса. Если один объект изменит значение такого поля, то это изменение увидят все объекты. Для работы со статическими атрибутами используются статические методы, объявленные со спецификатором **static**. Такие методы являются методами класса, не привязаны ни к какому объекту и не содержат указателя **this** на конкретный объект, вызвавший метод. Статические методы реализуют парадигму «раннего связывания», жестко определяющую версию метода на этапе компиляции. По причине недоступности указателя **this** статические поля и методы не могут обращаться к нестатическим полям и методам напрямую, так как для обращения к статическим полям и методам достаточно имени класса, в котором они определены.

```
// пример # 4 : статические метод и поле: Mark.java
package chapt03;

public class Mark {
      private int mark = 3;
      public static int coeff = 5;

      public double getResult() {
            return (double)coeff*mark/100;
      }
      public static void setCoeffFloat(float c) {
            coeff = (int)coeff*c;;
      }
      public void setMark(int mark) {
            this.mark = mark;
      }
//из статического метода нельзя обратиться к нестатическим полям и методам
      /*public static int getResult() {
            setMark(5);//ошибка
            return coeff*mark/100;//ошибка
      }*/
}
```

При создании двух объектов

```
Mark ob1 = new Mark();
Mark ob2 = new Mark();
```

Значение **ob1.coeff** и **ob2.coeff** и равно **5**, поскольку располагается в одной и той же области памяти. Изменить значение статического члена можно прямо через имя класса:

```
Mark.coeff = 7;
```

Вызов статического метода также следует осуществлять с помощью указания: **ClassName.*methodName*()**, а именно:

```
Mark.setCoeffFloat();
```

```
float z = Math.max(x, y); // определение максимума из двух значений
System.exit(1);  // экстренное завершение работы приложения
```

Статический метод можно вызывать также с использованием имени объекта, но такой вызов снижает качество кода и не будет логически корректным, хотя и не приведет к ошибке компиляции.

Переопределение статических методов класса не имеет практического смысла, так как обращение к статическому атрибуту или методу осуществляется по большей части посредством задания имени класса, которому они принадлежат.

Модификатор final

Модификатор **final** используется для определения констант в качестве члена класса, локальной переменной или параметра метода. Методы, объявленные как **final**, нельзя замещать в подклассах, для классов – создавать подклассы. Например:

```
/* пример # 5 : final-поля и методы: Rector.java: ProRector.java */
package chapt03;

public class Rector {

        // инициализированная константа
        final int ID = (int)(Math.random()*10);
        // неинициализированная константа
        final String NAME_RECTOR;

        public Rector() {
        // инициализация в конструкторе
                NAME_RECTOR = "Старый";// только один раз!!!
        }
        // {NAME_RECTOR = "Новый";} // только один раз!!!

        public final void jobRector() {
                // реализация
                // ID = 100; //ошибка!
        }
        public boolean checkRights(final int num) {
                // id = 1; //ошибка!
                final int CODE = 72173394;
                if (CODE == num) return true;
                else return false;
        }
        public static void main(String[] args) {
                System.out.println(new Rector().ID);
        }
}
package chapt03;

public class ProRector extends Rector {
        // public void jobRector(){} //запрещено!
}
```

Константа может быть объявлена как поле класса, но не проинициализирована. В этом случае она должна быть проинициализирована в логическом блоке класса, заключенном в **{}**, или конструкторе, но только в одном из указанных мест. Значение по умолчанию константа получить не может в отличие от переменных класса. Константы могут быть объявлены в методах как локальные или как параметры метода. В обоих случаях значения таких констант изменять нельзя.

Абстрактные методы

Абстрактные методы размещаются в абстрактных классах или интерфейсах, тела у таких методов отсутствуют и должны быть реализованы в подклассах.

```
/* пример # 6 : абстрактный класс и метод: AbstractCourse.java */
public abstract class AbstractCourse {
     private String name;
     public AbstractCourse()  { }
     public abstract void changeTeacher(int id);/*определение
                                          метода отсутствует*/
     public setName(String n){
            name = n;
     }
}
```

В отличие от интерфейсов, абстрактный класс может содержать и абстрактные, и неабстрактные методы, а может и не содержать ни одного абстрактного метода.

Подробнее абстрактные класса и интерфейсы изучаются в главе «Абстрактные классы. Интерфейсы. Пакеты».

Модификатор native

Приложение на языке Java может вызывать методы, написанные на языке C++. Такие методы объявляются с ключевым словом **native**, которое сообщает компилятору, что метод реализован в другом месте. Например:

```
public native int loadCripto(int num);
```

Методы, помеченные **native**, можно переопределять обычными методами в подклассах.

Модификатор synchronized

При использовании нескольких потоков управления в одном приложении необходимо синхронизировать методы, обращающиеся к общим данным. Когда интерпретатор обнаруживает **synchronized**, он включает код, блокирующий доступ к данным при запуске потока и снимающий блок при его завершении. Вызов методов уведомления о возвращении блокировки объекта **notifyAll()**, **notify()** и метода остановки потока **wait()** класса **Object** (суперкласса для всех классов языка Java) предполагает использование модификатора **synchronized**, так как эти методы предназначены для работы с потоками.

Логические блоки

При описании класса могут быть использованы логические блоки. Логическим блоком называется код, заключенный в фигурные скобки и не принадлежащий ни одному методу текущего класса, например:

```
{ /* код */ }
static { /* код */ }
```

Логические блоки чаще всего используются в качестве инициализаторов полей, но могут содержать вызовы методов и обращения к полям текущего класса. При создании объекта класса они вызываются последовательно, в порядке размещения, вместе с инициализацией полей как простая последовательность операторов, и только после выполнения последнего блока будет вызван конструктор класса. Операции с полями класса внутри логического блока до явного объявления этого поля возможны только при использовании ссылки **this**, представляющую собой ссылку на текущий объект.

Логический блок может быть объявлен со спецификатором **static**. В этом случае он вызывается только один раз в жизненном цикле приложения при создании объекта или при обращении к статическому методу (полю) данного класса.

```
/*пример # 7 : использование логических блоков при объявлении класса:
 Department.java: DemoLogic.java */
package chapt03;

public class Department {
      {
            System.out.println("logic (1) id=" + this.id);
      }
      static {
            System.out.println("static logic");
      }
      private int id = 7;

      public Department(int d) {
            id = d;
            System.out.println("конструктор id=" + id);
      }
      int getId() {
            return id;
      }
      {
            id = 10;
            System.out.println("logic (2) id=" + id);
      }
}
package chapt03;

public class DemoLogic {
```

```
    public static void main(String[] args) {
        Department obj = new Department(71);
        System.out.println("значение id=" + obj.getId());
    }
}
```

В результате выполнения этой программы будет выведено:

```
static logic
logic (1) id=0
logic (2) id=10
конструктор id=71
значение id=71
```

В первой строке вывода поле **id** получит значение по умолчанию, так как память для него выделена при создании объекта, а значение еще не проинициализировано. Во второй строке выводится значение **id**, равное **10**, так как после инициализации атрибута класса был вызван логический блок, изменивший его значение.

Перегрузка методов

Метод называется перегруженным, если существует несколько его версий с одним и тем же именем, но с различным списком параметров. Перегрузка реализует «раннее связывние». Перегрузка может ограничиваться одним классом. Методы с одинаковыми именами, но с различающимися списком параметров и возвращаемыми значениями могут находиться в разных классах одной цепочки наследования и также будут являться перегруженными. Если в последнем случае списки параметров совпадают, то имеет место другой механизм – переопределение метода.

Статические методы могут перегружаться нестатическими и наоборот – без ограничений.

При вызове перегруженных методов следует избегать ситуаций, когда компилятор будет не в состоянии выбрать тот или иной метод.

/ пример # 8 : вызов перегруженных методов: NumberInfo.java */*

```
package chapt04;

public class NumberInfo {
    public static void viewNum(Integer i) {//1
        System.out.printf("Integer=%d%n", i);
    }
    public static void viewNum(int i) {//2
        System.out.printf("int=%d%n", i);
    }
    public static void viewNum(Float f) {//3
        System.out.printf("Float=%.4f%n", f);
    }
    public static void viewNum(Number n) {//4
        System.out.println("Number=" + n);
    }
    public static void main(String[] args) {
```

```
            Number[] num =
                  {new Integer(7), 71, 3.14f, 7.2 };
            for (Number n : num)
                  viewNum(n);

            viewNum(new Integer(8));
            viewNum(81);
            viewNum(4.14f);
            viewNum(8.2);
      }
}
```

Может показаться, что в результате компиляции и выполнения данного кода будут последовательно вызваны все четыре метода, однако в консоль будет выведено:

```
Number=7
Number=71
Number=3.14
Number=7.2
Integer=8
int=81
Float=4,1400
Number=8.2
```

То есть во всех случаях при передаче в метод элементов массива был вызван четвертый метод. Это произошло вследствие того, что выбор варианта перегруженного метода происходит на этапе компиляции и зависит от типа массива **num**. То, что на этапе выполнения в метод передается другой тип (для первых трех элементов массива), не имеет никакого значения, так как выбор уже был осуществлен заранее.

При непосредственной передаче объекта в метод выбор производится в зависимости от типа ссылки на этапе компиляции.

С одной стороны, этот механизм снижает гибкость, с другой – все возможные ошибки при обращении к перегруженным методам отслеживаются на этапе компиляции, в отличие от переопределенных методов, когда их некорректный вызов приводит к возникновению исключений на этапе выполнения.

При перегрузке всегда следует придерживаться следующих правил:

- не использовать сложных вариантов перегрузки;
- не использовать перегрузку с одинаковым числом параметров;
- заменять при возможности перегруженные методы на несколько разных методов.

Параметризованные классы

К наиболее важным новшествам версии языка J2SE 5 можно отнести появление параметризованных (generic) классов и методов, позволяющих использовать более гибкую и в то же время достаточно строгую типизацию, что особенно важно при работе с коллекциями. Применение generic-классов для создания типизированных коллекций будет рассмотрено в главе «Коллекции». Параметризация позволяет создавать классы, интерфейсы и методы, в которых тип обрабатываемых данных задается как параметр.

Приведем пример generic-класса с двумя параметрами:

```
/*пример # 9 : объявление класса с двумя параметрами : Subject.java */
package chapt03;

public class Subject <T1, T2> {
      private T1 name;
      private T2 id;

      public Subject() {
      }
      public Subject(T2 ids, T1 names) {
            id = ids;
            name = names;
      }
}
```

Здесь **T1**, **T2** – фиктивные объектные типы, которые используются при объявлении членов класса и обрабатываемых данных. При создании объекта компилятор заменит все фиктивные типы на реальные и создаст соответствующий им объект. В качестве параметров классов запрещено применять базовые типы.

Объект класса **Subject** можно создать, например, следующим образом:

```
Subject<String,Integer> sub =
      new Subject<String,Integer>();
          char ch[] = {'J','a','v','a'};
Subject<char[],Double> sub2 =
                new Subject<char[],Double>(ch, 71D );
```

В объявлении **sub2** имеет место автоупаковка значения **71D** в **Double**.

Параметризированные типы обеспечивают типовую безопасность.

Ниже приведен пример параметризованного класса **Optional** с конструкторами и методами, также инициализация и исследование поведения объектов при задании различных параметров.

```
/*пример # 10 : создание и использование объектов параметризованного
 класса: Optional.java: Runner.java */
package chapt03;

public class Optional <T> {
      private T value;

      public Optional() {
      }
      public Optional(T value) {
            this.value = value;
      }
      public T getValue() {
            return value;
      }
      public void setValue(T val) {
```

```
            value = val;
      }
      public String toString() {
            if (value == null) return null;
          return value.getClass().getName() + " " + value;
      }
}
package chapt03;

public class Runner {
      public static void main(String[] args) {
      //параметризация типом Integer
            Optional<Integer> ob1 =
                        new Optional<Integer>();
            ob1.setValue(1);
            //ob1.setValue("2");// ошибка компиляции: недопустимый тип
            int v1 = ob1.getValue();
            System.out.println(v1);
      //параметризация типом String
            Optional<String> ob2 =
                        new Optional<String>("Java");
            String v2 = ob2.getValue();
            System.out.println(v2);
      //ob1 = ob2; //ошибка компиляции – параметризация не ковариантна

      //параметризация по умолчанию – Object
            Optional ob3 = new Optional();
            System.out.println(ob3.getValue());
            ob3.setValue("Java SE 6");
            System.out.println(ob3.toString());/* выводится
                        тип объекта, а не тип параметризации */

            ob3.setValue(71);
            System.out.println(ob3.toString());

            ob3.setValue(null);
      }
}
```

В результате выполнения этой программы будет выведено:

```
1
Java
null
java.lang.String Java SE 6
java.lang.Integer 71
```

В рассмотренном примере были созданы объекты типа **Optional**: **ob1** на основе типа **Integer** и **ob2** на основе типа **String** при помощи различных конструкторов. При компиляции вся информация о generic-типах стирается и

заменяется для членов класса и методов заданными типами или типом **Object**, если параметр не задан, как для объекта **ob3**. Такая реализация необходима для обеспечения совместимости с кодом, созданным в предыдущих версиях языка.

Объявление generic-типа в виде **<T>**, несмотря на возможность использовать любой тип в качестве параметра, ограничивает область применения разрабатываемого класса. Переменные такого типа могут вызывать только методы класса **Object**. Доступ к другим методам ограничивает компилятор, предупреждая возможные варианты возникновения ошибок.

Чтобы расширить возможности параметризованных членов класса, можно ввести ограничения на используемые типы при помощи следующего объявления класса:

```
public class OptionalExt <T extends Тип> {
     private T value;
}
```

Такая запись говорит о том, что в качестве типа **T** разрешено применять только классы, являющиеся наследниками (суперклассами) класса **Тип**, и соответственно появляется возможность вызова методов ограничивающих (bound) типов.

Часто возникает необходимость в метод параметризованного класса одного допустимого типа передать объект этого же класса, но параметризованного другим типом. В этом случае при определении метода следует применить метасимвол **?**. Метасимвол также может использоваться с ограничением **extends** для передаваемого типа.

```
/*пример # 11 : использование метасимвола в параметризованном классе:
Mark.java, Runner.java */
package chapt03;

public class Mark<T extends Number> {
      public T mark;

      public Mark(T value) {
            mark = value;
      }
      public T getMark() {
            return mark;
      }
      public int roundMark() {
            return Math.round(mark.floatValue());
      }
      /* вместо */ // public boolean sameAny(Mark<T> ob) {
      public boolean sameAny(Mark<?> ob) {
            return roundMark() == ob.roundMark();
      }
      public boolean same(Mark<T> ob) {
            return getMark() == ob.getMark();
      }
}
```

```
package chapt03;

public class Runner {
      public static void main(String[] args) {
      // Mark<String> ms = new Mark<String>("7"); //ошибка компиляции
            Mark<Double> md = new Mark<Double>(71.4D);//71.5d
            System.out.println(md.sameAny(md));
            Mark<Integer> mi = new Mark<Integer>(71);
            System.out.println(md.sameAny(mi));
            // md.same(mi); //ошибка компиляции
            System.out.println(md.roundMark());
      }
}
```

В результате будет выведено:

```
true
true
71
```

Метод **sameAny(Mark<?> ob)** может принимать объекты типа **Mark**, инициализированные любым из допустимых для этого класса типов, в то время как метод с параметром **Mark<T>** мог бы принимать объекты с инициализацией того же типа, что и вызывающий метод объект.

Для generic-типов существует целый ряд ограничений. Например, невозможно выполнить явный вызов конструктора generic-типа:

```
class Optional <T> {
      T value = new T();
}
```

так как компилятор не знает, какой конструктор может быть вызван и какой объем памяти должен быть выделен при создании объекта.

По аналогичным причинам generic-поля не могут быть статическими, статические методы не могут иметь generic-параметры или обращаться к generic-полям, например:

```
/*пример # 12 : неправильное объявление полей параметризованного класса:
Failed.java */
package chapt03;

class Failed <T1, T2> {
      static T1 value;
      T2 id;

      static T1 getValue() {
          return value;
        }
      static void use() {
          System.out.print(id);
       }
   }
```

Параметризованные методы

Параметризованный (generic) метод определяет базовый набор операций, которые будут применяться к разным типам данных, получаемых методом в качестве параметра, и может быть записан, например, в виде:

```
<T extends Тип> returnType methodName(T arg) {}
<T> returnType methodName(T arg) {}
```

Описание типа должно находиться перед возвращаемым типом. Запись первого вида означает, что в метод можно передавать объекты, типы которых являются подклассами класса, указанного после **extends**. Второй способ объявления метода никаких ограничений на передаваемый тип не ставит.

Generic-методы могут находиться как в параметризованных классах, так и в обычных. Параметр метода может не иметь никакого отношения к параметру своего класса. Метасимволы применимы и к generic-методам.

/ пример # 13 : параметризованный метод: GenericMethod.java */*

```
public class GenericMethod {
     public static <T extends Number> byte asByte(T num) {
             long n = num.longValue();
             if (n >= -128 && n <= 127) return (byte)n;
             else return 0;
     }
     public static void main(String [] args) {
          System.out.println(asByte(7));
          System.out.println(asByte(new Float("7.f")));
          // System.out.println(asByte(new Character('7'))); // ошибка компиляции
     }
}
```

Объекты типа **Integer** (**int** будет в него упакован) и **Float** являются подклассами абстрактного класса **Number**, поэтому компиляция проходит без затруднений. Класс **Character** не обладает вышеуказанным свойством, и его объект не может передаваться в метод **asByte(T num)**.

Методы с переменным числом параметров

Возможность передачи в метод нефиксированного числа параметров позволяет отказаться от предварительного создания массива объектов для его последующей передачи в метод.

/ пример # 14: определение количества параметров метода: DemoVarargs.java */*

```
package chapt03;

public class DemoVarargs {
      public static int getArgCount(Integer... args) {
             if (args.length == 0)
                   System.out.print("No arg=");
             for (int i : args)
                   System.out.print("arg:" + i + "  ");
             return args.length;
      }
```

```
      public static void main(String args[]) {
        System.out.println("N=" + getArgCount(7, 71, 555));
        Integer[] i = { 1, 2, 3, 4, 5, 6, 7 };
        System.out.println("N=" + getArgCount(i));
        System.out.println(getArgCount());
      }
}
```

В результате выполнения этой программы будет выведено:

```
arg:7  arg:71  arg:555  N=3
arg:1  arg:2  arg:3  arg:4  arg:5  arg:6  arg:7  N=7
No arg=0
```

В примере приведен простейший метод с переменным числом параметров. Метод **getArgCount()** выводит все переданные ему аргументы и возвращает их количество. При передаче параметров в метод из них автоматически создается массив. Второй вызов метода в примере позволяет передать в метод массив. Метод может быть вызван и без аргументов.

Чтобы передать несколько массивов в метод по ссылке, следует использовать следующее объявление:

```
void methodName(Тип[]... args){}
```

Методы с переменным числом аргументов могут быть перегружены:

```
void methodName(Integer...args) {}
void methodName(int x1, int x2) {}
void methodName(String...args) {}
```

В следующем примере приведены три перегруженных метода и несколько вариантов их вызова. Отличительной чертой является возможность метода с аргументом **Object... args** принимать не только объекты, но и массивы:

```
/* пример # 15 : передача массивов: DemoOverload.java */
package chapt03;

public class DemoOverload {
      public static void printArgCount(Object... args) {//1
      System.out.println("Object args: " + args.length);
      }
      public static void printArgCount(Integer[]...args){//2
      System.out.println("Integer[] args: " + args.length);
      }
      public static void printArgCount(int... args) {//3
            System.out.print("int args: " + +args.length);
      }
      public static void main(String[] args) {
            Integer[] i = { 1, 2, 3, 4, 5 };

            printArgCount(7, "No", true, null);
            printArgCount(i, i, i);
            printArgCount(i, 4, 71);
            printArgCount(i);//будет вызван метод 1
```

```
            printArgCount(5, 7);
            // printArgCount();//неопределенность!
      }
}
```

В результате будет выведено:

```
Object args: 4
Integer[] args: 3
Object args: 3
Object args: 5
int args: 2
```

При передаче в метод **printArgCount()** единичного массива **i** компилятор отдает предпочтение методу с параметром **Object... args**, так как имя массива является объектной ссылкой и потому указанный параметр будет ближайшим. Метод с параметром **Integer[]...args** не вызывается, так как ближайшей объектной ссылкой для него будет **Object[]...args**. Метод с параметром **Integer[]...args** будет вызван для единичного массива только в случае отсутствия метода с параметром **Object...args**.

При вызове метода без параметров возникает неопределенность из-за невозможности однозначного выбора.

Не существует также ограничений и на переопределение подобных методов.

Единственным ограничением является то, что параметр вида **Тип...args** должен быть последним в объявлении метода, например:

```
void methodName(Тип1 obj, Тип2... args) {}
```

Перечисления

Типобезопасные перечисления (typesafe enums) в Java представляют собой классы и являются подклассами абстрактного класса **java.lang.Enum**. При этом объекты перечисления инициализируются прямым объявлением без помощи оператора **new**. При инициализации хотя бы одного перечисления происходит инициализация всех без исключения оставшихся элементов данного перечисления.

В качестве простейшего применения перечисления можно рассмотреть следующий код:

```
/* пример # 16 : применение перечисления: SimpleUseEnum.java */
package chapt02;

enum Faculty {
      MMF, FPMI, GEO
}
public class SimpleUseEnum {
      public static void main(String args[]) {
            Faculty current;
            current = Faculty.GEO;
            switch (current) {
              case GEO:
                  System.out.print(current);
                  break;
              case MMF:
```

```
                System.out.print(current);
                break;
          // case LAW : System.out.print(current);//ошибка компиляции!
            default:
           System.out.print("вне case: " + current);
          }
    }
}
```

В операторах **case** используются константы без уточнения типа перечисления, так как его тип определен в **switch**.

Перечисление как подкласс класса **Enum** может содержать поля, конструкторы и методы, реализовывать интерфейсы. Каждый тип **enum** может использовать методы:

static enumType[] values() – возвращает массив, содержащий все элементы перечисления в порядке их объявления;

static T valueOf(Class<T> enumType, String arg) – возвращает элемент перечисления, соответствующий передаваемому типу и значению передаваемой строки;

static enumType valueOf(String arg) – возвращает элемент перечисления, соответствующий значению передаваемой строки;

int ordinal() – возвращает позицию элемента перечисления.

```
/* пример # 17 : объявление перечисления с методом : Shape.java */
package chapt02;

enum Shape {
      RECTANGLE, TRIANGLE, CIRCLE;
      public double square(double x, double y) {
            switch (this) {
            case RECTANGLE:
                  return x * y;
            case TRIANGLE:
                  return x * y / 2;
            case CIRCLE:
                  return Math.pow(x, 2) * Math.PI;
            }
            throw new EnumConstantNotPresentException(
                  this.getDeclaringClass(),this.name());
      }
}
```

```
/* пример # 18 : применение перечисления: Runner.java */
package chapt02;

public class Runner {
      public static void main(String args[]) {
            double x = 2, y = 3;
            Shape[] arr = Shape.values();
```

```
            for (Shape sh : arr)
                  System.out.printf("%10s = %5.2f%n",
                              sh, sh.square(x, y));
      }
}
```

В результате будет выведено:

```
RECTANGLE =  6,00
 TRIANGLE =  3,00
   CIRCLE = 12,57
```

Каждый из элементов перечисления в данном случае представляет собой в том числе и арифметическую операцию, ассоциированную с методом **square()**. Без **throw** данный код не будет компилироваться, так как компилятор не исключает появления неизвестного элемента. Данная инструкция позволяет указать на возможную ошибку при появлении необъявленной фигуры. Поэтому и при добавлении нового элемента необходимо добавлять соответствующий ему **case**.

```
/* пример # 19 : конструкторы и члены перечисления: DeanDemo.java */
package chapt02;

enum Dean {
   MMF("Бендер"), FPMI("Балаганов"), GEO("Козлевич");
      String name;

      Dean(String arg) {
            name = arg;
      }
      String getName() {
            return name;
      }
}
package chapt02;

public class DeanDemo {
  public static void main(String[] args) {
    Dean dn = Dean.valueOf("FPMI");
    System.out.print(dn.ordinal());
    System.out.println(" : " + dn + " : " + dn.getName());
      }
}
```

В результате будет выведено:

```
1 : FPMI : Балаганов
```

Однако на перечисления накладывается целый ряд ограничений.

Им запрещено:

- быть суперклассами;
- быть подклассами;
- быть абстрактными;
- создавать экземпляры, используя ключевое слово **new**.

Аннотации

Аннотации – мета-теги, которые добавляются к коду и применяются к объявлению пакетов, типов, конструкторов, методов, полей, параметров и локальным переменным. В результате можно задать зависимость, например, метода от другого метода. Аннотации позволяют избежать создания шаблонного кода во многих ситуациях, активируя утилиты для его генерации из аннотаций в исходном коде.

В следующем коде приведено объявление аннотации.

```
/* пример # 20 : многочленная аннотация : RequestForCustomer.java */
package chapt03;

public @interface RequestForCustomer {
      int level();
      String description();
      String date();
}
```

Ключевому слову **interface** предшествует символ **@**. Такая запись сообщает компилятору об объявлении аннотации. В объявлении также есть три метода-члена: **int level()**, **String description()**, **String date()**.

Все аннотации содержат только объявления методов, добавлять тела этим методам не нужно, так как их реализует сам язык. Кроме того, эти методы не могут содержать параметров, секции **throws** и действуют скорее как поля. Допустимые типы возвращаемого значения: базовые типы, **String**, **Enum**, **Class** и массив любого из вышеперечисленных типов.

Все типы аннотаций автоматически расширяют интерфейс **java.lang.annotation.Annotation**. В этом интерфейсе даны методы: **hashCode()**, **equals()** и **toString()**, определенные в типе **Object**. В нем также приведен метод **annotationType()**, который возвращает объект типа **Class**, представляющий вызывающую аннотацию.

После объявления аннотации ее можно использовать для аннотирования объявлений. Объявление любого типа может иметь аннотацию, связанную с ним. Например, можно снабжать примечаниями классы, методы, поля, параметры и константы типа **enum**. Даже к аннотации можно добавить аннотацию. Во всех случаях аннотация предшествует объявлению.

Применяя аннотацию, нужно задавать значения для ее методов-членов. Далее приведен фрагмент, в котором аннотация **RequestForCustomer** сопровождает объявление метода:

```
@RequestForCustomer (
   level = 2,
   description = "Enable time",
   date = "10/10/2007"
)
public void customerThroughTime() {
   //...
}
```

Данная аннотация связана с методом **customerThroughTime()**. За именем аннотации, начинающимся с символа @, следует заключенный в круглые скобки список инициализирующих значений для методов-членов. Для того чтобы передать значение методу-члену, имени этого метода присваивается значение. Таким образом, в приведенном фрагменте строка **"Enable time"** присваивается методу **description()**, члену аннотации типа **RequestForCustomer**. При этом в присваивании после имени **description** нет круглых скобок. Когда методу-члену передается инициализирующее значение, используется только имя метода. Следовательно, в данном контексте методы-члены выглядят как поля.

```
/* пример # 21 : применение аннотации: Request.java */
package chapt03;
import java.lang.reflect.Method;

public class Request {
      @RequestForCustomer(level = 2,
                          description = "Enable time",
                          date = "10/10/2007")
      public void customerThroughTime() {
            try {
            Class c = this.getClass();
            Method m = c.getMethod("customerThroughTime");
            RequestForCustomer ann =
            m.getAnnotation(RequestForCustomer.class);
                  //запрос аннотаций
                  System.out.println(ann.level() + " "
                              + ann.description() + " "
                              + ann.date());
            } catch (NoSuchMethodException e) {
                  System.out.println("метод не найден");
            }
      }
      public static void main(String[] args) {
            Request ob = new Request();
            ob.customerThroughTime();
      }

   }
```

В результате будет выведено:

```
2 Enable time 10/10/2007
```

Если аннотация объявляется в отдельном файле, то ей нужно задать правило сохранения **RUNTIME** в виде кода

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
@Retention(RetentionPolicy.RUNTIME) //правило сохранения
```

помещаемого перед объявлением аннотации, которое предоставляет максимальную продолжительность существования аннотации.

С правилом **SOURCE** аннотация существует только в исходном тексте программы и отбрасывается во время компиляции.

Аннотация с правилом сохранения **CLASS** помещается в процессе компиляции в файл **.class**, но не доступна в JVM во время выполнения.

Аннотация, заданная с правилом сохранения **RUNTIME**, помещается в файл **.class** в процессе компиляции и доступна в JVM во время выполнения. Следовательно, правило **RUNTIME** предлагает максимальную продолжительность существования аннотации.

Основные типы аннотаций: аннотация-маркер, одночленная и многочленная.

Аннотация-маркер не содержит методов-членов. Цель – пометить объявление. В этом случае достаточно присутствия аннотации. Поскольку у интерфейса аннотации-маркера нет методов-членов, достаточно определить наличие аннотации.

```
public @interface TigerAnnotation {}
```

Для проверки наличия аннотации-маркера используется метод **isAnnotationPresent()**.

Одночленная аннотация содержит единственный метод-член. Для этого типа аннотации допускается краткая условная форма задания значения для метода-члена. Если есть только один метод-член, то просто указывается его значение при создании аннотации. Имя метода-члена указывать не нужно. Но для того чтобы воспользоваться краткой формой, следует для метода-члена использовать имя **value()**.

Многочленные аннотации содержат несколько методов-членов. Поэтому используется полный синтаксис (**имя_параметра = значение**) для каждого параметра.

В языке Java определено семь типов встроенных аннотаций, четыре типа – **@Retention**, **@Documented**, **@Target** и **@Inherited** – импортируются из пакета **java.lang.annotation**. Оставшиеся три – **@Override**, **@Deprecated** и **@Suppresswarnings** – из пакета **java.lang**.

Аннотации получают все более широкое распространение и активно используются в различных технологиях.

Задания к главе 3

Вариант А

1. Определить класс **Вектор** размерности n. Реализовать методы сложения, вычитания, умножения, инкремента, декремента, индексирования. Определить массив из m объектов. Каждую из пар векторов передать в методы, возвращающие их скалярное произведение и длины. Вычислить и вывести углы между векторами.
2. Определить класс **Вектор** размерности n. Определить несколько конструкторов. Реализовать методы для вычисления модуля вектора, скалярного произведения, сложения, вычитания, умножения на константу. Объявить массив объектов. Написать метод, который для заданной пары векторов будет определять, являются ли они коллинеарными или ортогональными.

3. Определить класс **Вектор** в R^3. Реализовать методы для проверки векторов на ортогональность, проверки пересечения неортогональных векторов, сравнения векторов. Создать массив из `m` объектов. Определить, какие из векторов компланарны.
4. Определить класс **Матрица** размерности (`n` x `n`). Класс должен содержать несколько конструкторов. Реализовать методы для сложения, вычитания, умножения матриц. Объявить массив объектов. Создать методы, вычисляющие первую и вторую нормы матрицы $\| a \|_1 = \max\limits_{1 \le i \le n} \sum\limits_{j=1}^{n} (a_{ij}), \|a\|_2 = \max\limits_{1 \le j \le n} \sum\limits_{i=1}^{n} (a_{ij})$. Определить, какая из матриц имеет наименьшую первую и вторую нормы.
5. Определить класс **Матрица** размерности (`m` x `n`). Класс должен содержать несколько конструкторов. Объявить массив объектов. Передать объекты в метод, меняющий местами строки с максимальным и минимальным элементами `k`-го столбца. Создать метод, который изменяет `i`-ю матрицу путем возведения ее в квадрат.
6. Определить класс **Цепная дробь** $A = a_0 + \cfrac{x}{a_1 + \cfrac{x}{a_2 + \cfrac{x}{a_3 + \ldots}}}$ Определить методы сложения, вычитания, умножения, деления. Вычислить значение для заданного `n`, `x`, `a[n]`.
7. Определить класс **Дробь** в виде пары (`m`,`n`). Класс должен содержать несколько конструкторов. Реализовать методы для сложения, вычитания, умножения и деления дробей. Объявить массив из `k` дробей, ввести/вывести значения для массива дробей. Создать массив объектов и передать его в метод, который изменяет каждый элемент массива с четным индексом путем добавления следующего за ним элемента массива.
8. Определить класс **Комплекс**. Класс должен содержать несколько конструкторов. Реализовать методы для сложения, вычитания, умножения, деления, присваивания комплексных чисел. Создать два вектора размерности `n` из комплексных координат. Передать их в метод, который выполнит их сложение.
9. Определить класс **Квадратное уравнение**. Класс должен содержать несколько конструкторов. Реализовать методы для поиска корней, экстремумов, а также интервалов убывания/возрастания. Создать масссив объектов и определить наибольшие и наименьшие по значению корни.
10. Определить класс **Булева матрица** (**BoolMatrix**) размерности (`n` x `m`). Класс должен содержать несколько конструкторов. Реализовать методы для логического сложения (дизъюнкции), умножения и инверсии матриц. Реализовать методы для подсчета числа единиц в матрице и упорядочения строк в лексикографическом порядке.
11. Построить класс **Булев вектор** (**BoolVector**) размерности `n`. Определить несколько конструкторов. Реализовать методы для

выполнения поразрядных конъюнкции, дизъюнкции и отрицания векторов, а также подсчета числа единиц и нулей в векторе.

12. Определить класс **Множество символов** мощности `n`. Написать несколько конструкторов. Реализовать методы для определения принадлежности заданного элемента множеству; пересечения, объединения, разности двух множеств. Создать методы сложения, вычитания, умножения (пересечения), индексирования, присваивания. Создать массив объектов и передавать пары объектов в метод другого класса, который строит множество, состоящее из элементов, входящих только в одно из заданных множеств.
13. Определить класс **Полином** степени `n`. Создать методы для сложения и умножения объектов. Объявить массив из m полиномов и передать его в метод, вычисляющий сумму полиномов массива. Определить класс **Рациональный полином** с полем типа **Полином**. Определить метод для сложения: $R = \frac{p_1(x)}{Q_1(x)} + \frac{p_2(x)}{Q_2(x)}$ и методы для ввода/вывода.
14. Определить класс **Нелинейное уравнение** для двух переменных. Написать несколько конструкторов. Создать методы для сложения и умножения объектов. Реализовать метод определения корней методом биекции.
15. Определить класс **Определённый интеграл** с аналитически подынтегральной функцией. Написать несколько конструкторов. Создать методы для вычисления значения по формуле левых прямоугольников, по формуле правых прямоугольников, по формуле средних прямоугольников, по формуле трапеций, по формуле Симпсона (параболических трапеций).
16. Определить класс **Массив** с аналитически подынтегральной функцией. Написать несколько конструкторов. Создать методы сортировки: обменная сортировка (метод пузырька); обменная сортировка «Шейкер-сортировка», сортировка посредством выбора (метод простого выбора), сортировка вставками: метод хеширования (сортировка с вычислением адреса), сортировка вставками (метод простых вставок), сортировка бинарного слияния, сортировка Шелла (сортировка с убывающим шагом).
17. Построить класс **Дерево**. Определить несколько конструкторов. Реализовать методы для отображение статистики об общем числе вершин в дереве, имеющих нечетные индексы, имеющих четные индексы, имеющих индексы, превышающие задаваемый пользователем порог. Определить метод вычисление расстояния между вершинами.

Вариант В

Создать классы, спецификации которых приведены ниже. Определить конструкторы и методы **`set`**`Тип`**`()`**, **`get`**`Тип`**`()`**, **`toString()`**. Определить дополнительно методы в классе, создающем массив объектов. Задать критерий выбора данных и вывести эти данные на консоль.

1. **Student**: id, Фамилия, Имя, Отчество, Дата рождения, Адрес, Телефон, Факультет, Курс, Группа.

Создать массив объектов. Вывести:
a) список студентов заданного факультета;
b) списки студентов для каждого факультета и курса;
c) список студентов, родившихся после заданного года;
d) список учебной группы.

2. **Customer**: id, Фамилия, Имя, Отчество, Адрес, Номер кредитной карточки, Номер банковского счета.
Создать массив объектов. Вывести:
a) список покупателей в алфавитном порядке;
b) список покупателей, у которых номер кредитной карточки находится в заданном интервале.

3. **Patient**: id, Фамилия, Имя, Отчество, Адрес, Телефон, Номер медицинской карты, Диагноз.
Создать массив объектов. Вывести:
a) список пациентов, имеющих данный диагноз;
b) список пациентов, номер медицинской карты у которых находится в заданном интервале.

4. **Abiturient**: id, Фамилия, Имя, Отчество, Адрес, Телефон, Оценки.
Создать массив объектов. Вывести:
a) список абитуриентов, имеющих неудовлетворительные оценки;
b) список абитуриентов, средний балл у которых выше заданного;
c) выбрать заданное число n абитуриентов, имеющих самый высокий средний балл (вывести также полный список абитуриентов, имеющих полупроходной балл).

5. **Book**: id, Название, Автор(ы), Издательство, Год издания, Количество страниц, Цена, Переплет.
Создать массив объектов. Вывести:
a) список книг заданного автора;
b) список книг, выпущенных заданным издательством;
c) список книг, выпущенных после заданного года.

6. **House**: id, Номер квартиры, Площадь, Этаж, Количество комнат, Улица, Тип здания, Срок эксплуатации.
Создать массив объектов. Вывести:
a) список квартир, имеющих заданное число комнат;
b) список квартир, имеющих заданное число комнат и расположенных на этаже, который находится в заданном промежутке;
c) список квартир, имеющих площадь, превосходящую заданную.

7. **Phone**: id, Фамилия, Имя, Отчество, Адрес, Номер кредитной карточки, Дебет, Кредит, Время городских и междугородных разговоров.
Создать массив объектов. Вывести:
a) сведения об абонентах, у которых время внутригородских разговоров превышает заданное;
b) сведения об абонентах, которые пользовались междугородной связью;
c) сведения об абонентах в алфавитном порядке.

8. **Car**: id, Марка, Модель, Год выпуска, Цвет, Цена, Регистрационный номер.
   Создать массив объектов. Вывести:
   a) список автомобилей заданной марки;
   b) список автомобилей заданной модели, которые эксплуатируются больше n лет;
   c) список автомобилей заданного года выпуска, цена которых больше указанной.
9. **Product**: id, Наименование, UPC, Производитель, Цена, Срок хранения, Количество.
   Создать массив объектов. Вывести:
   a) список товаров для заданного наименования;
   b) список товаров для заданного наименования, цена которых не превосходит заданную;
   c) список товаров, срок хранения которых больше заданного.
10. **Train**: Пункт назначения, Номер поезда, Время отправления, Число мест (общих, купе, плацкарт, люкс).
    Создать массив объектов. Вывести:
    a) список поездов, следующих до заданного пункта назначения;
    b) список поездов, следующих до заданного пункта назначения и отправляющихся после заданного часа;
    c) список поездов, отправляющихся до заданного пункта назначения и имеющих общие места.
11. **Bus**: Фамилия и инициалы водителя, Номер автобуса, Номер маршрута, Марка, Год начала эксплуатации, Пробег.
    Создать массив объектов. Вывести:
    a) список автобусов для заданного номера маршрута;
    b) список автобусов, которые эксплуатируются больше 10 лет;
    c) список автобусов, пробег у которых больше 100000 км.
12. **Airlines**: Пункт назначения, Номер рейса, Тип самолета, Время вылета, Дни недели.
    Создать массив объектов. Вывести:
    a) список рейсов для заданного пункта назначения;
    b) список рейсов для заданного дня недели;
    c) список рейсов для заданного дня недели, время вылета для которых больше заданного.

Тестовые задания к главе 3

Вопрос 3.1.

Какие из ключевых слов могут быть использованы при объявлении конструктора?

1) `private;`
2) `final;`
3) `native;`
4) `abstract;`
5) `protected.`

Вопрос 3.2.

Как следует вызвать конструктор класса **Quest3**, чтобы в результате выполнения кода была выведена на консоль строка "Конструктор".

```
public class Quest3 {
  Quest3 (int i){ System.out.print("Конструктор");     }
    public static void main(String[] args){
      Quest3 s= new Quest3();
      //1
    }
 public Quest3()     {
      //2
    }
      {
      //3
    } }
```

1) вместо //1 написать `Quest3(1);`
2) вместо //2 написать `Quest3(1);`
3) вместо //3 написать **new** `Quest3(1);`
4) вместо //3 написать `Quest3(1).`

Вопрос 3.3.

Какие из следующих утверждений истинные?

1) nonstatic-метод не может быть вызван из статического метода;
2) static-метод не может быть вызван из нестатического метода;
3) private-метод не может быть вызван из другого метода этого класса;
4) final-метод не может быть статическим.

Вопрос 3.4.

Дан код:

```
public class Quest5 {
      {System.out.print("1");}
      static{System.out.print("2");}
      Quest5(){System.out.print("3");}
      public static void main(String[] args) {
            System.out.print("4");
      } }
```

В результате при компиляции и запуске будет выведено:

1) 1234;
2) 4;
3) 34;
4) 24;
5) 14.