Программирование на Java

Лекция 6. Объявление классов

20 января 2003 года

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>

Аннотация

Центральная тема курса – объявление классов, поскольку любое Java-приложение является набором классов.

Первый рассматриваемый вопрос – система разграничения доступа в Java. Описывается, зачем вообще нужно управление доступом в ОО-языке программирования, и как оно осуществляется в Java. Затем подробно рассматривается структура объявления заголовка класса и его тела, которое состоит из элементов (полей и методов), конструкторов и инициализаторов. Дополнительно описывается сигнатура метода main, с которого начинается работа Java-приложения, правила передачи параметров различных типов в методы, перегруженные методы.

Оглавление

Лекция 6. Объявление классов 1
1. Введение 1
2. Модификаторы доступа 2
2.1. Предназначение модификаторов доступа 2
2.2. Разграничение доступа в Java 5
3. Объявление классов 9
3.1. Заголовок класса 9
3.2. Тело класса 10
3.3. Объявление полей 11
3.4. Объявление методов 12
3.5. Объявление конструкторов 16
3.6. Инициализаторы 22
4. Дополнительные свойства классов 24
4.1. Метод main 24
4.2. Параметры методов 25
4.3. Перегруженные методы 27
5. Заключение 28
6. Контрольные вопросы 28

Лекция 6. Объявление классов

Содержание лекции.

1. Введение 1

2. Модификаторы доступа 2

2.1. Предназначение модификаторов доступа 2

2.2. Разграничение доступа в Java 5

3. Объявление классов 9

3.1. Заголовок класса 9

3.2. Тело класса 10

3.3. Объявление полей 11

3.4. Объявление методов 12

3.5. Объявление конструкторов 16

3.6. Инициализаторы 22

4. Дополнительные свойства классов 24

4.1. Метод main 24

4.2. Параметры методов 25

4.3. Перегруженные методы 27

5. Заключение 28

6. Контрольные вопросы 28

1. Введение

Объявление классов является центральной темой курса, поскольку любая программа на Java - это набор классов. Поскольку типы являются ключевой конструкцией языка, их структура довольно сложна, имеет много тонкостей. Поэтому эта тема разделена на две главы.

Эта глава начинается с продолжения темы прошлой главы - имена и доступ к именованным элементам языка. Необходимо рассмотреть механизм разграничения доступа в Java, как он устроен, для чего применяется. Затем будут описаны ключевые правила объявления классов.

Следующая глава подробно рассматривает особенности объектной модели Java. Вводится понятие интерфейса. Уточняются правила объявления классов, и описывается объявление интерфейса.

2. Модификаторы доступа

Во многих языках существуют права доступа, которые ограничивают возможность использования, например, переменной в классе. Например, легко представить два крайних вида прав доступа: это public, когда поле доступно из любой точки программы, и private, когда поле может быть использовано только внутри того класса, в котором оно объявлено.

Однако прежде чем переходить к подробному рассмотрению этих и других модификаторов доступа, необходимо внимательно разобраться, зачем они вообще нужны.

2.1. Предназначение модификаторов доступа

Существует весьма распространенное мнение, которое расценивает права доступа как некий элемент безопасности кода: мол, необходимо защищать классы от "неправильного" использования. Например, если в классе Human (человек) есть поле age (возраст человека), то какой-нибудь программист по злому умыслу или незнанию может установить этому полю отрицательное значение, после чего объект станет работать неправильным образом, могут появиться ошибки. Для защиты такого поля age необходимо объявить его private.

Такая точка зрения довольно распространена, однако нужно признать, что она далека от истины. Основной потребностью в разграничении прав доступа является обеспечение неотъемлемого свойства объектной модели - инкапсуляции, то есть, сокрытия реализации. Исправим пример таким образом, чтобы он корректно отражал предназначение модификаторов доступа. Итак, пусть в классе Human есть поле age целочисленного типа, и чтобы все желающие могли пользоваться этим полем, оно объявляется public.

```
public class Human {
   public int age;
}
```

Проходит время, и если в группу программистов, работающих над системой, входят десятки разработчиков, то логично предположить, что все или многие из них начнут использовать это поле.

Вдруг может возникнуть ситуация, что целочисленного типа данных уже недостаточно, и хотелось бы сменить тип поля на дробный. Однако если просто изменить int на double, то вскоре все разработчики, которые пользовались классом Human и его полем age, обнаружат, что в их коде появились ошибки, потому что поле вдруг стало дробным, и в строках, подобной этой:

```
Human h = getHuman();
int i=h.age; // Ошибка!!
```

будет возникать ошибка из-за попытки провести неявным образом сужение примитивного типа.

Получается, что подобное изменение (в общем, небольшое и локальное) потребует модификации многих и многих классов. Поэтому внесение его окажется недопустимым, неоправданным с точки зрения количества усилий, которые необходимо затратить. То есть, объявив один раз поле или метод как public, можно оказаться в ситуации, когда

малейшие изменения (имени, типа, характеристик, правил использования) в дальнейшем станут невозможны.

Напротив, если бы поле было объявлено как private, а для чтения и изменения его значения были бы введены дополнительные методы, то ситуация поменялась бы в корне:

```
public class Human {
   private int age;

   // метод, возвращающий значение age
   public int getAge() {
      return age;
 }

   // метод, устанавливающий значение age
   public void setAge(int a) {
      age=a;
   }
}
```

В этом случае с этим классом могло бы работать множество программистов, и могло бы быть создано большое количество классов, использующих тип Human, но модификатор private дает гарантию, что никто напрямую этим полем не пользуется, и изменение его типа было бы совершенно безболезненной операцией, связанной с изменением в ровно одном классе.

Получение величины возраста выглядело бы следующим образом:

```
Human h = getHuman();
int i=h.getAge(); // Обращение через метод
```

Рассмотрим, как выглядит процесс смены типа поля age:

```
public class Human {

   // поле получает новый тип double
   private /*int*/ double age;

   // старые методы работают с округлением значения
   public int getAge() {
      return (int)Math.round(age);
   }
   public void setAge(int a) {
      age=a;
   }

   // добавляются новые методы для работы с типом double
   public double getExactAge() {
      return age;
   }
```

```
   public void setExactAge(double a) {
      age=a;
   }

}
```

Видно, что старые методы, которые возможно уже применяются во множестве мест, остались без изменения. Точнее, остался без изменений их внешний формат, а внутренняя реализация усложнилась. Но такая перемена не потребует никаких модификаций остальных классов системы. Пример использования

```
Human h = getHuman();
int i=h.getAge(); // Корректно
```

остается верным, переменная i получает корректное целое значение. Однако изменения вводились для возможности работать с дробными величинами. Для этого были добавлены новые методы, и во всех местах, где требуется точное значение возраста, необходимо обращаться к ним:

```
Human h = getHuman();
double d=h.getExactAge(); // Точное значение возраста
```

Итак, в класс была добавлена новая возможность, не потребовавшая никаких изменений уже написанного кода.

За счет чего была достигнута такая гибкость? Необходимо выделить свойства объекта, которые необходимы будущим пользователям этого класса, и их сделать доступными (в данном случае, public). Те же элементы класса, что содержат детали внутренней реализации логики класса, желательно скрывать от внешнего мира, чтобы не образовались нежелательные зависимости, которые могут серьезно сдержать развитие системы.

Этот пример одновременно иллюстрирует и другое теоретическое правило написания объектов, а именно: в большинстве случаев доступ к полям лучше реализовывать через специальные методы (accessors) для чтения (getters) и записи (setters). То есть, само поле рассматривается как деталь внутренней реализации. Действительно, если рассматривать внешний интерфейс объекта как целиком состоящий из допустимых действий, то доступными элементами должны быть только методы, реализующие эти действия. Один из случаев, в котором такой подход приносит необходимую гибкость, уже рассмотрен.

Есть и другие соображения. Например, вернемся к вопросу о корректном использовании объекта и установки верных значений полям. Как следствие, правильное разграничение доступа позволяет ввести механизмы проверки входных значений:

```
public void setAge(int a) {
   (if a>=0) {
   age=a;
}
}
```

В этом примере поле age никогда не примет некорректное отрицательное значение. (Недостатком приведенного примера является то, что в случае неправильных входных данных, они просто игнорируется, нет никаких сообщений, позволяющих узнать, что изменения поля возраста на самом деле не произошло; для полноценной реализации метода необходимо освоить работу с ошибками в Java).

Бывают и более существенные изменения логики класса. Например, данные могут начать храниться не в полях класса, а в более надежном хранилище, например, файловой системе или базе данных. В этом случае методы-аксессоры опять изменят свою реализацию, и начнут обращаться к persistent storage (постоянное хранилище, например, БД) для чтения/записи значений. Если доступа к полям класса не было, а открытыми были только методы для работы с их значениями, то можно довольно легко изменить код этих методов, а наружные типы, которые использовали этот класс, совершенно не изменятся, логика их работы останется той же.

Подведем итоги. Функциональность класса необходимо разделять на открытый интерфейс, описывающий действия, которые будут использовать внешние типы, и на внутреннюю реализацию, которая используется только внутри самого класса. Внешний интерфейс в дальнейшем модифицировать невозможно или очень сложно для больших систем, поэтому его требуется продумывать особенно тщательно. Детали внутренней реализации могут быть изменены на любом этапе, если они не меняют логику работы всего класса. Благодаря такому подходу реализуется одна из базовых характеристик объектной модели - инкапсуляция, и обеспечивается важное преимущество технологии ООП - модульность.

Таким образом, модификаторы доступа вводятся не для защиты типа от внешнего пользователя, а, напротив, для защиты, или избавления, пользователя от излишних зависимостей от деталей внутренней реализации. Что же касается неправильного использования класса, то его создателям нужно стремится к тому, чтобы класс был легок и понятен в применении, тогда таких проблем не возникнет, ведь программист не станет сознательно писать код, который порождает ошибки в его программе.

Конечно, такое разбиение на внешний интерфейс и внутреннюю реализацию не всегда очевидно, часто условно. Для облегчения задачи технических дизайнеров классов в Java введено не 2 (public и private), а 4 уровня доступа. Рассмотрим их и весь механизм разграничения доступа в Java более подробно.

2.2. Разграничение доступа в Java

Уровень доступа элемента языка является статическим свойством, задается на уровне кода и всегда проверяется во время компиляции. Попытка обратиться к закрытому элементу вызовет ошибку.

В Java модификаторы доступа указываются для:

- типов (классов и интерфейсов) объявления верхнего уровня;
- элементов ссылочных типов (полей, методов, внутренних типов);
- конструкторов классов.

Как следствие, массив также может быть недоступен в том и только в том случае, если не доступен тип, на основе которого он объявлен.

Все 4 уровня доступа имеют только элементы типов и конструкторы. Это:

- public;
- private;
- protected;
- если не указан ни один из этих 3 типов, то уровень доступа определяется по умолчанию (default).

Первые два из них уже были рассмотрены. Последний уровень (доступ по умолчанию) упоминался в прошлой главе - он позволяет обращения из того же пакета, где объявлен и сам этот класс. По этой причине пакеты в Java являются не просто набором типов, а более структурированной единицей, так как типы внутри одного пакета могут больше взаимодействовать друг с другом, чем с типами из других пакетов.

Наконец, protected дает доступ наследникам класса. Понятно, что наследникам может потребоваться доступ к некоторым элементам родителя, с которыми не требуется иметь дело внешним классам.

Однако описанная структура не позволяет упорядочить модификаторы доступа так, чтобы каждый следующий строго расширял предыдущий. Модификатор protected может быть указан для наследника из другого пакета, а доступ по умолчанию позволяет обращения из классов-ненаследников, если они находятся в том же пакете. По этой причине возможности protected были расширены таким образом, что он включает в себя доступ внутри пакета. Итак, модификаторы доступа упорядочиваются следующим образом (от менее открытых - к более):

```
private
(none) default
protected
public
```

Эта последовательность будет использована далее при изучении деталей наследования классов.

Теперь рассмотрим, какие модификаторы доступа возможны для различных элементов языка.

- Пакеты всегда доступны, поэтому у них нет модификаторов доступа (можно сказать, что все они public, то есть, любой существующий в системе пакет может быть использован из любой точки программы).
- Типы (классы и интерфейсы) верхнего уровня объявления. При их объявлении есть всего две возможности: указать модификатор public или не указывать его. Если доступ к типу является public, то это означает, что он доступен из любой точки кода. Если же он не public, то уровень доступа назначается по умолчанию: тип доступен только внутри того пакета, где он объявлен.
- Массив имеет тот же уровень доступа, что и тип, на основе которого он объявлен (естественно, все примитивные типы являются полностью доступными).
- Элементы и конструкторы объектных типов. Обладают всеми 4 возможными значениями уровня доступа. Все элементы интерфейсов являются public.

Для типов объявления верхнего уровня нет необходимости во всех 4 уровнях доступа. private-типы образовывали бы закрытую мини-программу, никто не мог бы их использовать. Типы, доступные только для наследников, также не были признаны полезными.

Разграничения доступа сказываются не только на обращении к элементам объектных типов или пакетов (через составное имя или прямое обращение), но также при вызове конструкторов, наследовании, приведении типов. Запрещается импортировать недоступные типы.

Проверка уровня доступа проводится компилятором. Обратите внимание на следующие примеры:

```
public class Wheel {
   private double radius;

   public double getRadius() {
      return radius;
   }
}
```

Значение поля radius не доступно снаружи класса, однако открытый метод getRadius() корректно возвращает его.

Рассмотрим следующие два модуля компиляции:

```
package first;

// Некоторый класс Parent
public class Parent {
}
```

```
package first;

// Класс Child наследуется от класса Parent,
// но имеет ограничение доступа по умолчанию
class Child extends Parent {
}

public class Provider {
   public Parent getValue() {
      return new Child();
   }
}
```

К методу getValue() класса Provider можно обратиться и из другого пакета, не только из пакета first, поскольку метод объявлен как public. Как видно, этот метод возвращает экземпляр класса Child, который не доступен из других пакетов. Однако следующий вызов является корректным:

```
package second;
```

```
import first.*;

public class Test {
   public static void main(String s[]) {
      Provider pr = new Provider();
      Parent p = pr.getValue();
      System.out.println(p.getClass().getName());
      // (Child)p - приведет к ошибке компиляции!
   }
}
```

Результатом будет:

```
first.Child
```

То есть, на самом деле в классе Test работа идет с экземпляром недоступного класса Child, что возможно, поскольку обращение к нему делается через открытый класс Parent. Попытка же сделать явное приведение вызовет ошибку. Да, тип объекта "угадан" верно, но доступ к закрытому типу всегда запрещен.

Следующий пример:

```
public class Point {
   private int x, y;

   public boolean equals(Object o) {
      if (o instanceof Point) {
         Point p = (Point)o;
         return p.x==x && p.y==y;
      }
      return false;
   }
}
```

В этом примере объявляется класс Point с двумя полями, описывающими координаты точки. Обратите внимание, что поля полностью закрыты - private. Далее попытаемся переопределить стандартный метод equals() таким образом, чтобы для аргументов, являющихся экземплярами класса Point, или его наследников (логика работы оператора intsanceof) возвращалось истинное значение в случае равенства координат. Обратите внимание на строку, где делается сравнение координат - для этого приходится обращаться к private-полям другого объекта!

Тем не менее, такое действие совершенно корректно, поскольку private допускает обращения из любой точки класса, независимо от того, к какому именно объекту оно производится.

Другие примеры разграничения доступа в Java будут рассматриваться по ходу курса.

3. Объявление классов

Рассмотрим базовые возможности объявления классов.

Объявление класса состоит из заголовка и тела класса.

3.1. Заголовок класса

Вначале указываются модификаторы класса. Модификаторы доступа для класса уже обсуждались. Допустимым является public, либо его отсутствие - доступ по умолчанию.

Класс может быть объявлен как final. В этом случае не допускается создание наследников такого класса. На своей ветке наследования он является последним. Класс String и классы-обертки, например, являются final-классами.

После списка модификаторов указывается ключевое слово class, а затем имя класса - корректный Java-идентификатор. Таким образом, кратчайшим объявлением класса может являться такой модуль компиляции:

```
class A {}
```

Фигурные скобки обозначают тело класса, но о нем позже.

Указанный идентификатор становится простым именем класса. Полное составное имя класса строится из полного составного имени пакета, в котором он объявлен (если это не безымянный пакет), и простого имени класса, разделенных точкой. Область видимости класса, где он может быть доступен по своему простому имени - его пакет.

Далее заголовок может содержать ключевое слово extends, после которого должно быть указано имя (простое или составное) доступного не-final класса. В этом случае объявляемый класс наследуется от указанного класса. Если выражение extends не применяется, то класс наследуется напрямую от Object. Выражение extends Object допускается и игнорируется.

```
class Parent {} // = class Parent extends Object {}

final class LastChild extends Parent {}

// class WrongChild extends LastChild {} // ошибка!!
```

Попытка расширить final-класс приведет к ошибке компиляции.

Если в объявлении класса A указано выражение extends B, то класс A называют прямым наследником класса B.

Класс A считается наследником класса B если:

- A является прямым наследником B;
- либо существует класс C, который является наследником B, а A является наследником C (это правило применяется рекурсивно).

Таким образом можно проследовать цепочки наследования на несколько уровней вверх.

Если компилятор обнаруживает, что класс является своим наследником, то возникает ошибка компиляции:

```
// пример вызовет ошибку компиляции
class A extends B {}
class B extends C {}
class C extends A {} // ошибка! Класс A стал своим наследником
```

Далее в заголовке может быть указано ключевое слово implements, за которым должно следовать перечисление через запятую имен (простых или составных, повторения запрещены) доступных интерфейсов:

```
public final class String implements Serializable, Comparable {}
```

В этом случае говорят, что класс реализует перечисленные интерфейсы. Как видно из примера, класс может реализовывать любое количество интерфейсов. Если выражение implements отсутствует, то класс действительно не реализует никаких интерфейсов, здесь значений по умолчанию нет.

Далее следует пара фигурных скобок, которые могут быть пустыми или содержать описание тела класса.

3.2. Тело класса

Тело класса может содержать объявление элементов (members) класса:

- полей;
- методов;
- внутренних типов (классов и интерфейсов);

и остальных допустимых конструкций:

- конструкторов;
- инициализаторов;
- статических инициализаторов.

Элементы класса имеют имена и передаются по наследству, не-элементы - нет. Для элементов простые имена указываются при объявлении, составные формируются из имени класса или имени переменной объектного типа и простого имени элемента. Областью видимости элементов является все объявление тела класса. Допускается применение любого из всех 4 модификатора доступа. Напоминаем, что соглашения по именованию классов и их элементов обсуждались в прошлой главе.

Не-элементы не обладают именами, а потому не могут быть вызваны явно. Их вызывает сама виртуальная машина. Например, конструктор вызывается при создании объекта. По той же причине неэлементы не обладают модификаторами доступа.

Элементами класса являются элементы, описанные в объявлении тела класса и переданные по наследству от класса-родителя (кроме Object - единственного класса, не имеющего родителя) и всех реализуемых интерфейсов при условии достаточного уровня доступа. Таким образом, если класс содержит элементы с доступом по умолчанию, то его наследники из разных пакетов будут обладать разным набором элементов. Классы из того

же пакета могут пользоваться полным набором элементов, а из других пакетов - только protected и public. private-элементы не передаются по наследству.

Поля и методы могут иметь одинаковые имена, поскольку обращение к полям всегда записывается без скобок, а к методам - всегда со скобками.

Рассмотрим все эти конструкции более подробно.

3.3. Объявление полей

Объявление полей начинается с перечисления модификаторов. Возможно применение любого из 3 модификаторов доступа, либо никакого вовсе, что означает уровень доступа по умолчанию.

Поле может быть объявлено final, что означает, что оно инициализируется ровно один раз и больше не будет менять своего значения. Простейший способ работы с final-переменными - инициализация при объявлении:

```
final double PI=3.1415;
```

Также допускается инициализация final-полей в конце каждого конструктора класса.

Не обязательно использовать для инициализации константы компиляции, возможно обращение к различным функциям, например:

```
final long creationTime=System.currentTimeMillis();
```

Данное поле будет хранить время создания объекта. Существуют еще два специальных модификатора - transient и volatile. Они будут рассмотрены в соответствующих главах.

После списка модификаторов указывается тип поля. Затем идет перечисление одного или нескольких имен полей с возможными инициализаторами:

```
int a;
int b=3, c=b+5, d;
Point p, p1=null, p2=new Point();
```

Повторяющиеся имена полей запрещены. Указанный идентификатор при объявлении становится простым именем поля. Составное имя формируется из имени класса или имени переменной объектного типа и простого имени поля. Областью видимости поля является все объявление тела класса.

Запрещается использовать поле в инициализации других полей до его объявления.

```
int y=x;
int x=3;
```

Однако в остальном поля можно объявлять и ниже их использования:

```
class Point {
   int getX() {return x;}
```

```
   int y=getX();
   int x=3;

   public static void main (String s[]) {
      Point p=new Point();
      System.out.println(p.x+", "+p.y);
   }
}
```

Результатом будет:

```
3, 0
```

Данный пример корректен, но для понимания его результата необходимо вспомнить, что все поля класса имеют значение по умолчанию:

- для числовых полей примитивных типов это 0;
- для булевского типа это false;

Таким образом, при инициализации переменной y был использован результат метода getX(), который вернул значение по умолчанию переменной x, то есть 0. Затем переменная x получила значение 3.

3.4. Объявление методов

Объявление метода состоит из заголовка и тела метода. Заголовок состоит из:

- модификаторов (доступа в том числе);
- типа возвращаемого значения или ключевого слова void;
- имени метода;
- списка аргументов в круглых скобках (аргументов может не быть);
- специального throws-выражения.

Заголовок начинается с перечисления модификаторов. Для методов доступен любой из 3 возможных модификаторов доступа. Также допускается использование доступа по умолчанию.

Кроме этого, существует модификатор final, который говорит о том, что такой метод нельзя переопределять в наследниках. Можно считать, что все методы final-класса, а также все private-методы любого класса являются final.

Затем, поддерживается модификатор native. Метод, объявленный с таким модификатором, не имеет реализации на Java. Он должен быть написан на другом языке (C/C++, Fortran и т.д.) и добавлен в систему в виде загружаемой динамической библиотеки (например, DLL для Windows). Существует специальная спецификация JNI (Java Native Interface), описывающая правила создания и использования native-методов.

Такая возможность необходима для Java, поскольку многие компании имеют обширные программные библиотеки, написанные на более старых языках. Их было бы очень трудоемко и неэффективно переписывать на Java, поэтому необходима возможность подключать их в таком виде, в каком они есть. Безусловно, при этом Java-приложения теряют целый ряд

своих преимуществ, такие как переносимость, безопасность и другие. Поэтому применять JNI надо только в случае крайней необходимости.

Эта спецификация накладывает требования на имена процедур во внешних библиотеках (она составляет их из имени пакета, класса и самого native-метода), а поскольку библиотеки менять, как правило, очень неудобно, часто пишут специальные библиотеки-"обертки", к которым обращаются Java-классы через JNI, а они сами обращаются к целевым модулям, пришедшим из прошлого.

Наконец, существует еще один специальный модификатор synchronized, который будет рассмотрен в главе, описывающей потоки выполнения.

После перечисления модификаторов указывается имя (простое или составное) типа возвращаемого значения; это может быть как примитивный, так и объектный тип. Если метод не возвращает никакого значения, указывается ключевое слово void.

Затем определяется имя метода. Указанный идентификатор при объявлении становится простым именем метода. Составное имя формируется из имени класса или имени переменной объектного типа и простого имени метода. Областью видимости метода является все объявление тела класса.

Аргументы метода перечисляются через запятую. Для каждого указывается сначала тип, затем имя параметра. В отличие от объявления переменной здесь запрещается указывать два имени для одного типа:

```
// void calc (double x, y); - ошибка!
void calc (double x, double y);
```

Если аргументы отсутствуют, указываются пустые круглые скобки. Одноименные параметры запрещены. Создание локальных переменных в методе, с именами, совпадающими с именами параметров, запрещено. Для каждого аргумента можно указать ключевое слово final перед указанием его типа. В этом случае такой параметр не может менять своего значения в теле метода (формально говоря, участвовать в операции присвоения в качестве левого операнда).

```
public void process(int x, final double y) {
   x=x*x+Math.sqrt(x);
   // y=Math.sin(x); - так писать нельзя, т.к. y - final!
}
```

Как происходит изменение значений аргументов метода, рассматривается в конце этой главы.

Важным понятием является сигнатура (signature) метода. Сигнатура определяется именем метода и его аргументами (количеством, типом, порядком следования). Если для полей запрещается совпадение имен, то для методов в классе запрещено создание двух методов с одинаковыми сигнатурами.

Например,

```
class Point {
   void get() {}
```

```
   void get(int x) {}
   void get(int x, double y) {}
   void get(double x, int y) {}
}
```

Такой класс объявлен корректно. Следующие пары методов несовместимы друг с другом в одном классе:

```
void get() {}
int get() {}

void get(int x) {}
void get(int y) {}

public int get() {}
private int get() {}
```

В первом случае методы отличаются типом возвращаемого значения, которое, однако, не входит в определение сигнатуры. Стало быть, это два метода с одинаковыми сигнатурами, и они не могут одновременно появится в объявлении тела класса. Можно легко составить пример, который создал бы неразрешимую проблему для компилятора, если бы был допустим:

```
// пример вызовет ошибку компиляции
class Test {
   int get() {
      return 5;
   }
   Point get() {
      return new Point(3,5);
   }

   void print(int x) {
      System.out.println("it's int! "+x);
   }
   void print(Point p) {
      System.out.println("it's Point! "+p.x+", "+p.y);
   }

   public static void main (String s[]) {
      Test t = new Test();
      t.print(t.get()); // Двусмысленность!
   }
}
```

В классе определена запрещенная пара методов get() с одинаковыми сигнатурами и различными возвращаемыми значениями. Обратимся к выделенной строке в методе main, где возникает конфликтная ситуация, с которой компилятор не сможет справиться. Определены два метода print() (у них разные аргументы, а значит и сигнатуры, то есть это

допустимые методы), и чтобы разобраться, какой из них будет вызван, нужно знать точный тип возвращаемого значения метода get(), что невозможно.

На основе этого примера можно понять, как составлено понятие сигнатуры. Действительно, при вызове указывается имя метода и перечисляются его аргументы, причем компилятор всегда может определить их тип. Как раз эти понятия и составляют сигнатуру, и требование ее уникальности позволяет компилятору всегда однозначно определить, какой метод будет вызван.

Аналогично, в предыдущем примере вторая пара методов различается именем аргументов, которые также не входят в определение сигнатуры и делают невозможным определение, какой их двух методов должен быть вызван.

Аналогично, третья пара различается лишь модификаторами доступа, что также недопустимо.

Наконец, завершает заголовок метода throws-выражение. Оно применяется для корректной работы с ошибками в Java и будет подробно рассмотрено в соответствующей главе.

Пример объявления метода:

```
public final java.awt.Point createPositivePoint(int x, int y)
   throws IllegalArgumentException
{
   return (x>0 && y>0) ? new Point(x, y) : null;
}
```

Далее, после заголовка метода следует тело метода. Оно может быть пустым, и тогда записывается одним символом "точка с запятой". native-методы всегда имеют только пустое тело, поскольку настоящая реализация написана на другом языке.

Обычные же методы имеют непустое тело, которое описывается в фигурных скобках, что можно видеть в многочисленных примерах в этой и других главах. Если текущая реализация метода не выполняет никаких действий, тело все равно должно описываться парой пустых фигурных скобок:

```
public void empty() {}
```

Если в заголовке метода указан тип возвращаемого значения, а не void, то в теле метода обязательно должно встречаться return-выражение. При этом компилятор проводит анализ структуры метода, чтобы гарантировать, что при любых операторах ветвления возвращаемое значение будет сгенерировано. Например, следующий пример является некорректным:

```
// пример вызовет ошибку компиляции
public int get() {
   if (condition) {
      return 5;
   }
}
```

Видно, что хотя тело метода содержит return-выражение, однако не при любом развитии событий возвращаемое значение будет сгенерировано. А вот такой пример является правильным:

```
public int get() {
   if (condition) {
      return 5;
   } else {
      return 3;
   }
}
```

Конечно, значение, указанное после слова return, должно быть совместимое по типу с объявленным возвращаемым значением (это понятие подробно рассматривается в главе "Преобразование типов").

В методе без возвращаемого значения (указано void) также можно использовать выражение return без каких либо аргументов. Его можно указать в любом месте метода, в этой точке выполнение метода будет завершено:

```
public void calculate(int x, int y) {
   if (x<=0 || y<=0) {
      return; // некорректные входные значения, выход из метода
   }
   ... // основные вычисления
}
```

Выражений return (с параметром или без для методов с/без возвращаемого значения) в теле одного метода может быть сколько угодно. Однако, следует помнить, что множество точек выхода в одном методе может серьезно усложнить понимание логики его работы.

3.5. Объявление конструкторов

Формат объявления конструкторов похож на упрощенное объявление методов. Также выделяют заголовок и тело конструктора. Заголовок состоит, во-первых, из модификаторов доступа (никакие другие модификаторы не допустимы). Затем указывается имя класса, которое можно расценивать двояко. Можно считать, что имя конструктора совпадает с именем класса. А можно рассматривать конструктор как безымянный, а имя класса - как тип возвращаемого значения, ведь конструктор может породить только объект класса, в котором он объявлен. Это исключительно дело вкуса, так как на формате объявления никак не сказывается:

```
public class Human {
   private int age;

   protected Human(int a) {
      age=a;
   }
```

```
   public Human(String name, Human mother, Human father) {
      age=0;
   }
}
```

Как видно из примеров, далее следует перечисление входных аргументов по тем же правилам, что и для методов. Также завершает заголовок конструктора throws-выражение. Оно имеет особую важность для конструкторов, поскольку создать ошибку - это единственный способ для конструктора не создавать объект. Если конструктор выполнился без ошибок, то объект гарантированно создается.

Тело конструктора пустым быть не может и поэтому всегда описывается в фигурных скобках (для простейших реализаций скобки могут быть пустыми).

В отсутствие имени (или, что то же самое, из-за того, что у всех конструкторов одинаковое имя, совпадающее с именем класса) сигнатура конструктора определяется только набором входных параметров по тем же правилам, что и для методов. Аналогично, в одном классе допускается любое количество конструкторов, если у них различные сигнатуры.

Тело конструктора может содержать любое количество return-выражений без аргументов. Если процесс исполнения дойдет до такого выражения, то на этом месте выполнение конструктора будет завершено.

Однако логика работы конструкторов имеет и некоторые важные особенности. Поскольку при их вызове осуществляется создание и инициал»изация объекта, то становится понятно, что такой процесс не может происходить без обращения к конструкторам всех родительских классов. Поэтому вводится обязательное правило - первой строкой в конструкторе должно быть обращение к родительскому классу, которое записывается с помощью ключевого слова super.

```
public class Parent {
   private int x, y;

   public Parent() {
      x=y=0;
   }

   public Parent(int newx, int newy) {
      x=newx;
      y=newy;
   }
}

public class Child extends Parent {
   public Child() {
      super();
   }

   public Child(int newx, int newy) {
      super(newx, newy);
```

```
    }
}
```

Как видно, обращение к родительскому конструктору записывается с помощью super, за которым идет перечисление аргументов. Этот набор определяет, какой из родительских конструкторов будет использован. В приведенном примере в каждом классе есть по 2 конструктора, и каждый конструктор в наследнике обращается к аналогичному в родителе (это довольно распространенный, но, конечно, не обязательный способ).

Проследим мысленно весь алгоритм создания объекта. Он начинается при исполнении выражения с ключевым словом new, за которым следует имя класса, от которого будет порождаться объект, и набор аргументов для его конструктора. По этому набору определяется, какой именно конструктор будет использован, и происходит его вызов. Первая строка его тела содержит вызов родительского конструктора. В свою очередь, первая строка тела конструктора родителя будет содержать вызов далее к его родителю и так далее. Восхождение по дереву наследования заканчивается, очевидно, на классе Object, у которого есть единственный конструктор без параметров. Его тело пустое (в смысле, записывается парой пустых фигурных скобок), однако можно считать, что именно в этот момент JVM порождает объект, и далее начинается процесс его инициализации. Выполнение начинает обратный путь вниз по дереву наследования. У самого верхнего родителя, прямого наследника от Object, происходит продолжение исполнения конструктора со второй строки. Когда он будет полностью выполнен, необходимо перейти к следующему родителю на один уровень наследования вниз и завершить выполнение его конструктора и так далее. Наконец, можно будет вернуться к конструктору исходного класса, который был вызван с помощью new, и также продолжить его выполнение со второй строки. По его завершению объект считается полностью созданным, исполнение выражения new будет закончено, а в качестве результата будет возвращена ссылка на порожденный объект.

Проиллюстрируем этот алгоритм следующим примером:

```
public class GraphicElement {
   private int x, y; // положение на экране

   public GraphicElement(int nx, int ny) {
      super(); // обращение к конструктору родителя Object
      System.out.println("GraphicElement");
      x=nx;
      y=nx;
   }
}

public class Square extends GraphicElement {
   private int side;

   public Square(int x, int y, int nside) {
      super(x, y);
      System.out.println("Square");
       side=nside;
   }
}
```

```
public class SmallColorSquare extends Square {
   private Color color;

   public SmallColorSquare(int x, int y, Color c) {
      super(x, y, 5);
      System.out.println("SmallColorSquare");
      color=c;
   }
}
```

После выполнения выражения создания объекта на экране появится следующее:

```
GraphicElement
Square
SmallColorSquare
```

Выражение super может стоять только на первой строке конструктора. Часто можно увидеть конструкторы вообще без такого выражения. В этом случае компилятор первой строкой по умолчанию добавляет вызов родительского конструктора без параметров (super()). Если у родительского класса нет такого конструктора, выражение super обязательно должно быть записано явно (и именно на первой строке), поскольку необходима передача входных параметров.

Напомним, что, во-первых, конструкторы не имеют имени и их нельзя вызвать явно, только через выражение создания объекта. Кроме того, конструкторы не передаются по наследству. То есть, если в родительском классе объявлено пять разных полезных конструкторов, и требуется, чтобы класс-наследник имел аналогичный набор, необходимо их все заново описать.

Класс обязательно должен иметь конструктор, иначе невозможно порождать объекты ни от него, ни от его наследников. Поэтому если в классе не объявлен ни один конструктор, компилятор добавляет один по умолчанию. Это public-конструктор без параметров и с телом, описанным парой пустых фигурных скобок. Из этого следует, что такое возможно только для классов, у родителей которых объявлен конструктор без параметров, иначе возникнет ошибка компиляции. Обратите внимание, что если затем в такой класс добавляется конструктор (не важно, с параметрами или без), то конструктор по умолчанию больше не вставляется:

```
/*
 * Этот класс имеет один конструктор.
 */
public class One {
   // Будет создан конструктор по умолчанию
   // Родительский класс Object имеет
   // конструктор без параметров.
}

/*
```

```
 * Этот класс имеет один конструктор.
 */
public class Two {
   // Единственный конструктор класса Second.
   // Выражение new Second() ошибочно!
   public Second(int x) {
   }
}

/*
 * Этот класс имеет два конструктора.
 */
public class Three extends Two {
   public Three() {
      super(1); // выражение super требуется
   }

   public Three(int x) {
      super(x); // выражение super требуется
   }
}
```

В случае если класс имеет более одного конструктора, допускается в первой строке некоторых из них указывать не super, а this - выражение, вызывающее другой конструктор этого же класса.

Рассмотрим следующий пример:

```
public class Vector {
   private int vx, vy;
   protected double length;

   public Vector(int x, int y) {
      super();
      vx=x;
      vy=y;
      length=Math.sqrt(vx*vx+vy*vy);
   }

   public Vector(int x1, int y1, int x2, int y2) {
      super();
      vx=x2-x1;
      vy=y2-y1;
      length=Math.sqrt(vx*vx+vy*vy);
   }
}
```

Видно, что оба конструктора совершают практически идентичные действия, поэтому можно применить более компактный вид записи:

```
public class Vector {
   private int vx, vy;
   protected double length;

   public Vector(int x, int y) {
      super();
      vx=x;
      vy=y;
      length=Math.sqrt(vx*vx+vy*vy);
   }

   public Vector(int x1, int y1, int x2, int y2) {
      this(x2-x1, y2-y1);
   }
}
```

Большим достоинством такого метода записи является то, что удалось избежать дублирования идентичного кода. Например, если процесс инициализации объектов этого класса удлинится на один шаг (скажем, добавится проверка длины на ноль), то такое изменение надо будет внести только в одно место - в первый конструктор. Такой подход помогает избегать случайных ошибок, так как исчезает необходимость тиражировать изменения в нескольких местах.

Разумеется, такое обращение к конструкторам своего класса не должно приводить к зацикливаниям, иначе будет выдана ошибка компиляции. Цепочка this должна в итоге приводить к super, который должен присутствовать (явно или неявно) хотя бы в одном из конструкторов. После того, как отработают конструкторы всех родительских классов, будет продолжено выполнение каждого конструктора, вовлеченного в процесс создания объекта.

После выполнения выражения new Test(0) на консоли появится:

```
Test()
Test(int x)
```

В заключение рассмотрим применение модификаторов доступа для конструкторов. Может вызвать удивление возможность объявлять конструкторы как private. Ведь они нужны для порождения объектов, а к таким конструкторам ни у кого не будет доступа. Однако в ряде случаев модификатор private может быть полезен. Например:

- private-конструктор может содержать инициализирующие действия, а остальные конструкторы будут использовать его с помощью this, причем прямое обращение к этому конструктору по каким-то причинам нежелательно;
- запрет на создание объектов этого класса, например, невозможно создать экземпляр класса Math;
- реализация специального шаблона проектирования из ООП Singleton, для работы которого требуется контролировать создание объектов, что невозможно в случае наличия не-private конструкторов.

3.6. Инициализаторы

Наконец, последней допустимой конструкцией в теле класса является объявление инициализаторов. Записываются объектные инициализаторы очень просто - внутри фигурных скобок.

```
public class Test {
   private int x, y, z;

   // инициализатор объекта
   {
      x=3;
      if (x>0)
         y=4;
      z=Math.max(x, y);
   }
}
```

Инициализаторы не имеют имен, исполняются при создании объектов и не могут быть вызваны явно, не передаются по наследству (хотя, конечно, инициализаторы в родительском классе продолжают исполняться при создании объекта класса-наследника).

Было указано уже три вида инициализирующего кода в классах - конструкторы, инициализаторы переменных, а теперь добавились объектные инициализаторы. Необходимо разобраться в какой последовательности что выполняется, в том числе при наследовании. При создании экземпляра класса вызванный конструктор выполняется следующим образом:

- если первой строкой идет обращение к конструктору родительского класса (явное или добавленное компилятором по умолчанию), то этот конструктор исполняется;
- в случае успешного исполнения вызываются все инициализаторы полей и объекта в том порядке, в каком они объявлены в теле класса;
- если первой строкой идет обращение к другому конструктору этого же класса, то он вызывается. Повторное выполнение инициализаторов не производится.

Второй пункт имеет ряд важных следствий. Во-первых, из него следует, что в инициализаторах нельзя использовать переменные класса, если их объявление записано позже.

Во-вторых, теперь можно сформулировать наиболее гибкий подход к инициализации final-полей. Главное требование - чтобы такие поля были проинициализированы ровно один раз. Это можно обеспечить в следующих случаях:

- инициализировать поле при объявлении;
- инициализировать поле ровно один раз в инициализаторе объекта (он должен быть записан после объявления поля);
- инициализировать поле ровно один раз в каждом конструкторе, в первой строке которого стоит явное или неявное обращение к конструктору родителя. Конструктор, в первой строке которого стоит this, не может и не должен инициализировать final-поле, так как

цепочка this-вызовов приведет к конструктору с super, в котором эта инициализация обязательно присутствует.

Для иллюстрации порядка исполнения инициализирующих конструкций рассмотрим следующий пример:

```
public class Test {
   {
      System.out.println("initializer");
   }

   int x, y=getY();
   final int z;

   {
      System.out.println("initializer2");
   }

   private int getY() {
      System.out.println("getY() "+z);
      return z;
   }
   public Test() {
      System.out.println("Test()");
      z=3;
   }
   public Test(int x) {
      this();
      System.out.println("Test(int)");
      // z=4; - нельзя! final-поле уже было инициализировано
   }
}
```

После выполнения выражения new Test() на консоли появится:

```
initializer
getY() 0
initializer2
Test()
```

Обратите внимание, что для инициализации поля y вызывается метод getY(), который возвращает значение final-поля z, которое в свою очередь еще не было инициализировано. Поэтому в итоге поле y получит значение по умолчанию 0, а затем поле z получит постоянное значение 3, которое никогда уже не изменится.

После выполнения выражения new Test(3) на консоли появится:

```
initializer
getY() 0
initializer2
```

```
Test()
Test(int)
```

4. Дополнительные свойства классов

Рассмотрим в этом разделе некоторые особенности работы с классами в Java. Обсуждение этого вопроса продолжится в специальной главе, посвященной объектной модели в Java.

4.1. Метод main

Итак, виртуальная машина реализуется приложением операционной системы и запускается по обычным правилам. Программа, написанная на Java, является набором классов. Понятно, что требуется некая входная точка, с которой должно начинаться выполнение приложения.

Такой входной точкой, по аналогии с языками C/C++, является метод main(). Пример его объявления:

```
public static void main(String[] args) {
}
```

Модификатор static в этой главе не рассматривался и будет изучен позже. Он позволяет вызвать метод main(), не создавая объектов. Метод не возвращает никакого значения, хотя в C есть возможность указать код возврата из программы. В Java для этой цели есть метод System.exit(), который закрывает виртуальную машину и имеет аргумент типа int.

Аргументом метода main() является массив строк. Он заполняется дополнительными параметрами, которые были указаны при вызове метода.

```
package test.first;

public class Test {
   public static void main(String[] args) {
      for (int i=0; i<args.length; i++) {
         System.out.print(args[i]+" ");
      }
      System.out.println();
   }
}
```

Для вызова программы виртуальной машине передается в качестве параметра имя класса, у которого объявлен метод main(). Поскольку это имя класса, а не имя файла, то не должно быть указано никакого расширения (.class или .java), а расположение класса записывается через точку (разделитель имен пакетов), а не с помощью файлового разделителя. Компилятору же, напротив, передается именно имя и путь к файлу.

Если вышеприведенный модуль компиляции сохранен в файле Test.java, который лежит в директории test\first, то вызов компилятора записывается следующим образом:

```
javac test\first\Test.java
```

А вызов виртуальной машины:

```
java test.first.Test
```

Чтобы проиллюстрировать работу с параметрами, изменим строку запуска приложения:

```
java test.first.Test Hello, World!
```

Результатом работы программы будет:

```
Hello, World!
```

4.2. Параметры методов

Для лучшего понимания работы с параметрами методов в Java необходимо рассмотреть несколько вопросов.

Во-первых, как передаются аргументы в методы - по значению или по ссылке? С точки зрения программы вопрос формулируется, например, следующим образом. Пусть есть переменная, и она в качестве аргумента передается в некоторый метод. Могут ли произойти какие-либо изменения с этой переменной после завершения работы метода?

```
int x=3;
process(x);
print(x);
```

Предположим, используемый метод объявлен следующим образом:

```
public void process(int x) {
   x=5;
}
```

Какое значение появится на консоли после выполнения примера? Чтобы ответить на этот вопрос необходимо вспомнить, как переменные разных типов хранят свои значения в Java.

Напомним, что примитивные переменные являются истинными хранилищами своих значений, и изменения значения одной переменной никогда не скажется на значении другой. Параметр метода process(), хоть и имеет такое же имя x, на самом деле является полноценным хранилищем целочисленной величины. А потому присвоение ему значения 5 не скажется на внешних переменных. То есть, результатом примера будет 3, и аргументы примитивного типа передаются в методы по значению. Единственный способ изменить такую переменную в результате работы метода - возвращать нужные величины из метода и использовать их при присвоении:

```
public int double(int x) {
   return x+x;
}
```

```
public void test() {
   int x=3;
   x=double(x);
}
```

Перейдем к ссылочным типам.

```
public void process(Point p) {
   p.x=3;
}

public void test() {
   Point p = new Point(1,2);
   process(p);
   print(p.x);
}
```

Ссылочная переменная хранит ссылку на объект, находящийся в памяти виртуальной машины. Поэтому аргумент метода process() будет иметь своим значением ту же самую ссылку и, стало быть, ссылаться на тот же самый объект. Изменения состояния объекта, осуществленные с помощью одной ссылки, всегда видны при обращении к этому объекту с помощью другой. Поэтому результатом примера будет значение 3. Объектные значения передаются в Java по ссылке.

Однако если изменять не состояние объекта, а саму ссылку, то результат будет другим:

```
public void process(Point p) {
   p = new Point(4,5);
}

public void test() {
   Point p = new Point(1,2);
   process(p);
   print(p.x);
}
```

В этом примере аргумент метода process() после присвоения начинает ссылаться на другой объект, нежели исходная переменная p, а значит, результатом примера станет значение 1. Можно сказать, что ссылочные величины передаются по значению, но значением является именно ссылка на объект.

Теперь можно уточнить, что означает возможность объявлять параметры методов и конструкторов как final. Поскольку изменения значений параметров (но не объектов, на которые они ссылаются) никак не сказываются на переменных вне метода, то модификатор final говорит лишь о том, что значение этого параметра не будет меняться на протяжении работы метода. Разумеется, для аргумента final Point p выражение p.x=5 является допустимым (запрещается p=new Point(5, 5)).

4.3. Перегруженные методы

Перегруженными (overloading) методами называются методы одного класса с одинаковыми именами. Сигнатуры у них должны быть различными, и различие может быть только в наборе аргументов.

Если в классе параметры перегруженных методов заметно различаются, например, у одного метода один параметр, у другого - два, то для Java это совершенно независимые методы, и совпадение их имен может служить только для повышения наглядности работы класса. Каждый вызов в зависимости от количества параметров однозначно адресуется к тому или иному методу.

Однако, если количество параметров одинаковое, а типы их различаются незначительно, но при вызове может сложиться двойственная ситуация, когда несколько перегруженных методов одинаково хорошо подходят для использования. Например, если объявлены типы Parent и Child, где Child расширяет Parent, то для следующих двух методов:

```
void process(Parent p, Child c) {}
void process(Child c, Parent p) {}
```

можно сказать, что они допустимы, их сигнатуры различаются. Однако при вызове

```
process(new Child(), new Child());
```

обнаруживается, что оба метода одинаково годятся для использования. Другой пример, методы:

```
process(Object o) {}
process(String s) {}
```

и примеры вызовов:

```
process(new Object());
process(new Point(4,5));
process("abc");
```

Легко видеть, что для первых двух вызовов подходящим является только первый метод, и именно он будет вызван. Для последнего же вызова подходят оба перегруженных метода, однако класс String является более "специфичным", или узким, чем класс Object. Действительно, значения типа String можно передавать в качестве аргументов типа Object, обратное же неверно. Компилятор попытается отыскать наиболее специфичный метод, подходящий для указанных параметров, и вызовет именно его. Поэтому при третьем вызове будет использован второй метод.

Однако для предыдущего примера такой подход не дает однозначного ответа. Оба метода одинаково специфичны для указанного вызова, и поэтому возникнет ошибка компиляции. Необходимо, используя явное приведение, указать компилятору, какой метод необходимо использовать:

```
process((Parent)(new Child()), new Child());
```

```
// или
process(new Child(),(Parent)(new Child()));
```

Это верно и в случае использования значения null:

```
process((Parent)null, null);
// или
process(null,(Parent)null);
```

5. Заключение

В этой главе началось рассмотрение ключевой конструкции языка Java – объявление класса.

Первая тема посвящена средствам разграничения доступа. Главный вопрос – для чего этот механизм вводится в практически каждом современном языке высокого уровня. Необходимо понимать, что он предназначен не для обеспечения «безопасности» или «защиты» объекта от неких неправильных действий. Самая важная задача – разделить внешний интерфейс класса и детали его реализации с тем, чтобы в дальнейшем воспользоваться такими преимуществами ООП, как инкапсуляция и модульность.

Затем были рассмотрены все 4 модификатора доступа, а также возможность их применения для различных элементов языка. Проверка уровня доступа проверяется уже на момент компиляции и запрещает лишь явное использование типов. Например, с ними все же можно работать через их более открытых наследников.

Объявление класса состоит из заголовка и тела класса. Формат заголовка был подробно описан. Для изучения тела класса необходимо вспомнить понятие элементов (members) класса. Ими могут быть поля, методы и внутренние типы. Для методов важным понятием является сигнатура.

Кроме того, в теле класса объявляются конструкторы и инициализаторы. Поскольку они не являются элементами, к ним нельзя обратиться явно, они вызываются самой виртуальной машиной. Также они не передаются по наследству.

Дополнительно был рассмотрен метод main, который вызывается при старте виртуальной машины. Далее описываются тонкости, возникающие при передаче параметров, и связанный с этим вопрос о перегруженных методах.

Рассмотрение классов в Java будет продолжено в следующих главах.

6. Контрольные вопросы

6-1. Какие модификаторы позволяют обращаться к элементу из классов того же пакета?

a.) Модификаторы public и protected, а также доступ по умолчанию (без указания модификатора).

6-2. Если в классе заводится новый элемент, и пока нет никаких факторов, позволяющих выбрать тот или иной модификатор доступа. Какой модификатор использовать в таком случае?

а.) Лучше использовать private. Раз не требуется внести этот элемент в открытый интерфейс класса, значит, стоит его отнести к реализации. В дальнейшем при необходимости всегда можно расширить уровень доступа. Обратное действие – сужение – запрещено.

6-3. Пусть класс User описывает пользователя системы. В качестве имени используется его e-mail адрес, который всем известен, а пароль, конечно, не должен быть никому доступен, кроме самого пользователя. Корректна ли следующая реализация?

```
public class User {
   public String login; // открытый e-mail
   private String password; // закрытый пароль
}
```

а.) Хотя код корректен с точки зрения компилятора, он не верен с точки зрения ООП дизайна. В предложенном варианте любой класс может изменить значение login у пользователя, что вряд ли соответствует задуманному алгоритму работы системы. Рекомендуется закрыть доступ к полю login извне и добавить метод для чтения, который бы возвращал значение login. Например:

```
public class User {
   private String login; // открытый e-mail
   private String password; // закрытый пароль

   public String getLogin(){
      return login;
   }
}
```

6-4. Корректен ли следующий код?

```
public class Test {
   private int id;

   public Test(int i) {
      id=i;
   }

   public static boolean test(Test t, int id) {
      return t.id==id;
   }
}
```

а.) Да. Метод test является методом класса Test, а значит, имеет доступ к полю id любого объекта этого класса.

6-5. Из каких частей состоит заголовок объявления класса? Тело класса?

a.) Заголовок класса (именно в таком порядке):

1. опциональные модификаторы (public, abstract, final)
2. ключевое слово class и имя класса
3. опционально ключевое слово extends и имя суперкласса
4. опционально ключевое слово implements и список имен реализуемых интерфейсов
5. тело класса в фигурных скобках

Тело класса (в произвольном порядке):

- поля
- методы
- внутренние типы (классы и интерфейсы)
- конструкторы
- инициализаторы

6-6. Если метод использует переменную класса, должна ли она быть объявлена выше объявления метода?

a.) Нет. Областью видимости переменной класса является все объявление тела класса.

6-7. Из каких частей состоит заголовок объявления метода? Какие части обязательные?

a.) Заголовок метода (именно в таком порядке):

1. опциональные модификаторы (доступа public|private|protected и прочие static, final, native, synchronized)
2. тип возвращаемого значения или void, если его нет
3. имя метода
4. список типов и имен аргументов в круглых скобках
5. опционально throws выражение

Обязательно должны присутствовать имя метода, тип возвращаемого значения (или void), перечисление аргументов в круглых скобках (или пустые скобки, если их нет).

6-8. Пусть класс должен обладать методом со следующими вариациями. Метод должен принимать в качестве аргумента дробное значение типа double или float и возвращать результат округления. В случае float нужно возвращать либо «короткое» значение (byte), либо «полное» (int). Аналогично для double – int или long соответственно. Сколько методов должно быть объявлено в таком классе, и каковы их сигнатуры?

a.) Если формально следовать вопросу, то должно быть объявлено 4 метода:

```
byte round(float x) { ... }
int roundToInt(float x) { ... }
```

```
int round(double x) { ... }
long roundToLong(double x) { ... }
```

Как видно, методы имеют различные имена, поскольку два из них отличаются только возвращаемыми значениями, а объявлять несколько методов с одинаковыми сигнатурами нельзя.

Однако можно заметить, что сигнатуры 2 и 3 методов очень близки, поэтому их можно объединить. Но и здесь нужна осторожность. Если просто убрать второй метод, то при вызове, например, round(1f) будет вызываться 1 метод, а не третий, как ожидалось. Поэтому объединение возможно одним из двух способов. Нужно либо для округления от float к int записывать вызов как round((double)1f), либо назвать 3 метод уникальным именем, чтобы он не был перегружен с первым.

6-9. Может ли класс не иметь ни одного конструктора?

a.) Нет, такой класс объявить невозможно. Даже если не указать ни одного конструктора, компилятор добавит конструктор по умолчанию. Если возникнет противоречие с родительским классом (ведь конструктор по умолчанию требует наличие доступного конструктора без параметров у родительского класса), то возникнет ошибка, и класс не будет скомпилирован.

6-10. Что появится на консоли при вызове конструктора следующего класса?

```
class Test {
   private long id=getId();
   private String name=getName();
   private String login;

   public Test(int domain) {
      login=domain+" "+name;
      System.out.println(login);
   }

   private static long getId() {
      int id = 3;
      System.out.println(id);
      return id;
   }

   private String getName() {
      String name="name"+getId();
      System.out.println(name);
      return name;
   }
}
```

а.) Результатом будет:

```
3
3
name3
5 name3
```

Первой будет проинициализирована переменная id (первая 3 на консоли). Затем начнет инициализироваться поле name, которое в свою очередь опять вызовет метод getId() (вторая 3), после чего напечатает name3. Затем в самом конструкторе распечатается значение 5 name3.

6-11. Как записывается заголовок метода main?

а.) `public static void main(String[] args)`

6-12. Может ли изменится содержимое переменной типа String, если передать ее в качестве аргумента при вызове метода?

а.) Нет, так как класс String, а стало быть, и его объекты, не изменяемы.