Лекция 4. Объявление классов

Введение

Объявление классов является центральной темой курса, поскольку любая программа на Java – это набор классов. Поскольку классы являются ключевой конструкцией языка, их структура довольно сложна, имеет много тонкостей.

Объявление классов

Рассмотрим базовые возможности объявления классов.

Объявление класса состоит из заголовка и тела класса.

Заголовок класса

Вначале указываются модификаторы класса. 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, представляют собой final -классы.

После списка модификаторов указывается ключевое слово class, а затем имя класса – корректный Java-идентификатор. Таким образом, кратчайшим объявлением класса может быть 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 ;
- существует класс C, который является наследником B, а A является наследником C (это правило применяется рекурсивно).

Таким образом можно проследить цепочки наследования на несколько уровней вверх.

Если компилятор обнаруживает, что класс является своим наследником, 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.

Тело класса

Тело класса может содержать объявление элементов (members) класса:

- полей;
- 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 четырех модификаторов доступа. Напоминаем, что соглашения по именованию классов и их элементов обсуждались в прошлой лекции.

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 не-элементы 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.

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.

Объявление полей

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

Поле может быть объявлено как final, это означает, что оно инициализируется 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. Существует еще два специальных модификатора - transient и volatile. Они будут рассмотрены в соответствующих лекциях.

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 – 0 ;
- для булевского типа – false ;
- для ссылочных – null.

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.

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 трех возможных модификаторов доступа. Также допускается использование доступа по умолчанию.

Кроме того, 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 - методылюбого класса, являются final.

Также 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.

Безусловно, при этом Java-приложения теряют целый ряд своих преимуществ, таких, как переносимость, безопасность и другие. Поэтому применять JNI следует 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.

Наконец, существует еще один специальный модификатор synchronized, который будет рассмотрен в лекции, 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 ввести ключевое слово final перед указанием его типа. В этом случае такой параметр не может менять своего значения в теле метода (то есть 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!
}
```

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 в одном классе друг с другом несовместимы:

```
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 появиться в объявлении тела класса.

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

Пример объявления метода:

```
public final 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 показано в многочисленных примерах в этой и других лекциях. 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 верным:

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

Конечно, значение, указанное после слова return, должно быть совместимо по типу с объявленным возвращаемым значением.

В методе без возвращаемого значения (указано void) также можно использовать выражение return без каких-либо аргументов. Его можно указать в любом месте метода и 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 заметно усложнить понимание логики его работы.

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 недопустимы). Во-вторых, 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. Завершает заголовок конструктора throws-выражение. Оно имеет особую важность для конструкторов, поскольку сгенерировать ошибку – это для конструктора единственный способ 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 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 инициализация объекта, 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 имеется по два 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 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 (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 завершении 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=ny;
      }
}

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 такого конструктора нет, 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 все их описать заново.

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 Two.
```

```
   // Выражение new Two() ошибочно!
   public Two(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
   }
}
```

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 увеличится на один шаг (скажем, добавится проверка длины на равенство нулю), то такое изменение надо будет внести только в первый конструктор. Такой подход помогает избежать 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.

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

   public Test(int x) {
      this();
      System.out.println("Test(int x)");
   }
}
```

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

```
Test()
```

```
Test(int x)
```

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, 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:

- если инициализировать поле при объявлении;
- если инициализировать поле только один раз в инициализаторе объекта (он должен быть записан после объявления поля);
- если инициализировать поле только 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 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)
```