Программирование на Java

Лекция 5. Имена. Пакеты

20 января 2003

Авторы документа:

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

Аннотация

В этой лекции рассматривается две темы – система именования элементов языка в Java и пакеты (packages), которые являются аналогами библиотек из других языков. Почти все конструкции в Java имеют имя для обращения к ним из других частей программы. По ходу изложения вводятся важные понятия, в частности – область видимости имени. При перекрытии таких областей возникает конфликт имен. Для того, чтобы минимизировать риск появления таких ситуаций, описываются соглашения по именованию, предложенные компанией Sun.

Пакеты осуществляют физическую и логическую группировку классов, и становятся необходимыми при создании больших систем. Вводится важное понятие модуля компиляции и описывается его структура.

Оглавление

Лекция 5. Имена. Пакеты

Содержание лекции.

1. Введение

Имена (names) используются в программе для доступа к объявленным (declared) ранее "объектам", "элементам", "конструкциям" языка (все эти слова-синонимы были использованы здесь в их общем смысле, а не как термины ООП, например). Конкретнее, в Java имеют имена:

- пакеты;
- классы;
- интерфейсы;
- элементы (member) ссылочных типов:
  - поля;
  - методы;
  - внутренние классы и интерфейсы;
- аргументы:
  - методов;
  - конструкторов;
  - обработчиков ошибок;
- локальные переменные.

Соответственно, все они должны быть объявлены специальным образом, что будет постепенно рассматриваться по ходу курса. Кроме этого, также объявляются конструкторы, однако их имя совпадает с именем класса, поэтому они не попали в этот список.

Напомним, что пакеты (packages) в Java - это способ логически группировать классы, что необходимо, поскольку зачастую количество классов в системе составляет несколько тысяч или даже десятков тысяч. Кроме классов и интерфейсов в пакетах могут находиться вложенные пакеты. Синонимами этого слова в других языках являются библиотека или модуль.

2. Имена

2.1. Простые и составные имена. Элементы.

Имена бывают простыми (simple), состоящими из одного идентификатора (они определяются во время объявления), и составными (qualified), состоящими из последовательности идентификаторов, разделенных точкой. Для пояснения этих терминов необходимо рассмотреть еще одно понятие.

У пакетов и ссылочных типов (классов, интерфейсов, массивов) есть элементы (members). Доступ к элементам осуществляется с помощью выражения, состоящего из имен, например, пакета и класса, разделенных точкой.

Далее классы и интерфейсы будут называться объединяющим термином тип (type).

Элементами пакета являются классы и интерфейсы, содержащиеся в этом пакете, а также вложенные пакеты. Чтобы получить составное имя пакета, необходимо к полному имени пакета, в котором он располагается, добавить точку, а затем его собственное простое имя. Например, составное имя основного пакета языка Java - java.lang (то есть, простое имя этого пакета lang, и он находится в объемлющем пакете java). Внутри него есть вложенный пакет, предназначенный для типов технологии reflection, которая упоминалась в предыдущих главах. Простое название пакета reflect, а, значит, составное - java.lang.reflect.

Простое имя классов и интерфейсов дается при объявлении, например Object, String, Point. Чтобы получить составное имя таких типов надо к составному имени пакета, в котором находится тип, через точку добавить простое имя типа. Например, java.lang.Object, java.lang.reflect.Method или com.myfirm.MainClass. Смысл последнего выражения таков: сначала идет обращение к пакету com, затем к его элементу - вложенному пакету myfirm, а затем к элементу пакета myfirm - классу MainClass. Здесь com.myfirm - составное имя пакета, где лежит класс MainClass, а MainClass -простое имя этого класса. Составляем их и разделяем точкой - получается полное имя класса com.myfirm.MainClass.

Для ссылочных типов элементами являются поля и методы, а также внутренние типы (классы и интерфейсы). Элементы могут быть как непосредственно объявлены в классе, так и получены по наследству от родительских классов и интерфейсов, если таковые имеются. Простое имя элементов также дается при инициализации. Например, toString(), PI, InnerClass. Составное имя получается путем объединения простого или составного имени типа или переменной объектного типа с именем элемента. Например, ref.toString(), java.lang.Math.PI, OuterClass.InnerClass. Другие обращения к элементам ссылочных типов уже неоднократно применялись в примерах в предыдущих главах.

2.2. Имена и идентификаторы

Теперь, когда были рассмотрены простые и составные имена, уточним разницу между идентификатором (напомним, что это вид лексемы) и именем. Понятно, что простое имя

состоит из одного идентификатора, а составное - из нескольких. Однако не всякий идентификатор входит в состав имени.

Во-первых, в выражении объявления (declaration) идентификатор еще не является именем. Другими словами, идентификатор становится именем после первого появления в коде в месте объявления.

Во-вторых, есть возможность обращаться к полям и методам объектного типа не через имя типа или объектной переменной, а через ссылку на объект, полученную в результате выполнения выражения. Один пример такого случая уже приводился в предыдущих главах:

```
country.getCity().getStreet();
```

В данном примере getStreet является не именем, а идентификатором, так как соответствующий метод вызывается у объекта, полученного в результате вызова метода getCity(). Причем, country.getCity - как раз является составным именем метода.

Наконец, идентификаторы также используются для названий меток (label). Эта конструкция рассматривается позже, однако приведем пример, иллюстрирующий, что пространства имен и названий меток полностью разделены.

```
num:
   for (int num = 2; num <= 100; num++) {
      int n = (int)Math.sqrt(num)+1;
      while (--n != 0) {
         if (num%n==0) {
            continue num;
         }
      }
      System.out.print(num+" ");
   }
```

Результатом будут простые числа, меньшие 100:

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Легко видеть, что применяются одноименные переменная и метка num, причем последняя используется для выхода из внутреннего цикла while на внешний for.

Очевидно, что удобнее использовать простое имя, а не составное, т.к. оно короче и его легче запомнить. Однако понятно, что если в системе есть очень много классов, у которых много переменных, то можно столкнуться с ситуацией, когда в разных классах есть одноименные переменные или методы. Для решения этой и других подобных проблем вводится новое понятие - область видимости.

2.3. Область видимости (введение)

Чтобы не заставлять программистов, совместно работающих над различными классами одной системы, координировать имена, которые они дают различным конструкциям языка, у каждого имени есть область видимости (scope). Если обращение к, например, полю идет

из части кода, попадающей в область видимости его имени, то можно пользоваться простым именем, если нет - то необходимо применять составное.

Например,

```
class Point {
   int x,y;

   int getX() {
      return x; // простое имя
   }
}

class Test {
   void main() {
      Point p = new Point();
      p.x=3; // составное имя
   }
}
```

Видно, что к полю x изнутри класса можно обращаться по простому имени. К нему же из другого класса можно обратиться только по составному имени. Оно составляется из имени переменной, ссылающейся на объект, и имени поля.

Теперь необходимо рассмотреть области видимости для всех элементов языка. Однако прежде познакомимся ближе с тем, что такое пакеты, как и для чего они используются.

3. Пакеты

Программа на Java представляет собой набор пакетов (packages). Каждый пакет может включать вложенные пакеты, то есть они образуют иерархическую систему.

Кроме этого, пакеты могут содержать классы и интерфейсы, и таким образом группируют типы, что необходимо сразу для нескольких целей. Во-первых, чисто физически невозможно работать с большим количеством классов, если они "свалены в кучу". Во-вторых, модульная декомпозиция облегчает проектирование системы. К тому же, как будет рассмотрено ниже, существует специальный уровень доступа, позволяющий типам из одного пакета более "тесно" взаимодействовать друг с другом, чем с классами из других пакетов. Таким образом, с помощью пакетов производится логическая группировка типов. Из ООП известно, что большая связность системы, то есть среднее количество классов, с которыми взаимодействует каждый класс, серьезно усложняет развитие и поддержку такой системы. С применением пакетов гораздо проще эффективно организовать взаимодействие подсистем друг с другом.

Наконец, каждый пакет имеет свое пространство имен, что позволяет создавать одноименные классы в различных пакетах. Таким образом, разработчикам не приходится тратить время на разрешение конфликта имен.

3.1. Элементы пакета

Еще раз повторим, что элементами пакета являются вложенные пакеты и типы (классы и интерфейсы). Одноименные элементы запрещены, то есть не может быть одноименных класса и интерфейса или вложенного пакета и типа. В противном случае возникнет ошибка компиляции.

Например, в JDK 1.0 пакет java содержал следующие пакеты: applet, awt, io, lang, net, util; и не содержал ни одного типа. Пакет java.awt содержал вложенный пакет image и 46 классов и интерфейсов.

Составное имя любого элемента пакета составляется из составного имени этого пакета и простого имени элемента. Например, для класса Object в пакете java.lang составным именем будет java.lang.Object, а для пакета image в пакете java.awt - java.awt.image.

Иерархическая структура пакетов была введена для удобства организации связанных пакетов, однако вложенные пакеты или соседние, то есть вложенные в один и тот же пакет, не имеют никаких дополнительных связей между собой, кроме ограничения на несовпадение имен. Например, пакеты space.sun, space.sun.ray, space.moon и factory.store совершенно "равны" между собой, и типы из одного из этих пакетов не имеют никакого особенного доступа к типам других пакетов.

3.2. Платформенная поддержка пакетов

Простейшим способом организации пакетов и типов является обычная файловая структура. Рассмотрим вырожденный пример, когда все пакеты, исходный и бинарный код располагаются в одной директории и ее поддиректориях.

В этой корневой директории должна быть папка java, соответствующая основному пакету языка, а в ней, в свою очередь, вложенные папки applet, awt, io, lang, net, util.

Предположим, разработчик работает над моделью солнечной системы, для чего создал классы Sun, Moon и Test, и расположил их в пакете space.sunsystem. В таком случае в корневой директории будет папка space, которая будет соответствовать одноименному пакету, а в ней - папка sunsystem, в которой хранятся классы этого разработчика.

Как известно, исходный код располагается в файлах с расширением .java, а бинарный - с расширением .class. Таким образом, содержимое папки sunsystem может выглядеть следующим образом:

```
Moon.java
Moon.class
Sun.java
Sun.class
Test.java
Test.class
```

Другими словами, исходный код классов

```
space.sunsystem.Moon
space.sunsystem.Sun
space.sunsystem.Test
```

хранится в файлах

```
space\sunsystem\Moon.java
space\sunsystem\Sun.java
space\sunsystem\Test.java
```

а бинарный код - в соответствующих .class-файлах. Обратите внимание, что преобразование имен пакетов в файловые пути потребовало замены разделителя . (точки) на символ-разделитель файлов (для Windows это обратный слеш \). Такое преобразование легко сможет сделать как компилятор для поиска исходных текстов и бинарного кода, так и виртуальная машина для загрузки классов и интерфейсов.

Обратите внимание, что было бы ошибкой запускать Java прямо из папки space\sunsystem и пытаться обращаться к классу Test, несмотря на то, что файл-описание лежит именно в ней. Необходимо подняться на два уровня директорий выше, чтобы Java, построив путь из имени пакета, смогла обнаружить нужный файл.

Кроме того, немаловажно то, что Java всегда различает регистр идентификаторов, а значит, и названия файлов и директорий должны точно отвечать запрограммированным именам. Хотя в некоторых случаях операционная система может обеспечить доступ, не взирая на регистр, но при изменении обстоятельств расхождения могут привести к сбоям.

Существует специальное выражение, объявляющее пакет (подробно рассматривается ниже). Оно предшествует объявлению типа и обозначает, какому пакету будет принадлежать этот тип. Таким образом, набор доступных пакетов определяется набором доступных файлов, содержащих объявления типов и пакетов. Например, если создать пустую директорию, или заполнить ее посторонними файлами, то это отнюдь не приведет к появлению пакета в Java.

Какие файлы доступны для утилит Java SDK (компилятора, интерпретатора и т.д.), устанавливается на уровне операционной системы, ведь утилиты - это обычные программы, которые выполняются под управлением ОС, и, конечно, следуют ее правилам. Например, если пакет содержит один тип, но файл-описание этого типа недоступен для чтения текущему пользователю ОС, то для Java этот тип и этот пакет не будут существовать.

Понятно, что далеко не всегда удобно хранить все файлы в одной директории. Зачастую разные классы находятся в разных местах, а некоторые могут даже распространятся в виде архивов, для ускорения загрузки через сеть. Требование копировать все такие файлы в одну папку было бы крайне затруднительным.

Поэтому Java использует специальную переменную окружения, которая называется classpath. Аналогично тому, как переменная path помогает системе находить и загружать динамические библиотеки, эта переменная помогает работать с Java-классами. Ее значение должно состоять из путей к директориям или архивам, разделенных точкой с запятой. С версии 1.1 поддерживаются архивы типов ZIP и JAR (Java ARchive) - специальный формат, разработанный на основе ZIP для Java.

Например, переменная classpath может иметь такое значение:

```
.;c:\java\classes;d:\lib\3Dengine.zip;d:\lib\fire.jar
```

В результате все указанные директории и содержимое всех архивов "добавляется" к исходной корневой директории. Java в поисках класса будет искать его по описанному

выше правилу во всех указанных папках и архивах по порядку. Обратите внимание, что первым в переменной указана текущая директория (представлена точкой). Это делается для того, чтобы поиск всегда начинался с исходной корневой директории. Конечно, такая запись не является обязательной и делается на усмотрение разработчика.

Несмотря на явные удобства такой конструкции, она таит в себе и опасности. Если при работе случилось так, что разрабатываемые классы лежат в некоторой директории, и она указана в classpath позже, чем некая другая директория, в которой обнаруживаются одноименные типы, то разобраться в такой ситуации будет непросто. В классы будут вноситься изменения, которые никак не проявляются при запуске из-за того, что Java на самом деле загружает одни и те же файлы из посторонней папки.

Поэтому к этой переменной среды окружения необходимо относится с вниманием. Полезно помнить, что необязательно устанавливать ее значение сразу для всей операционной системы. Его можно явно указывать при каждом запуске компилятора или виртуальной машины как опцию, что, во-первых, никогда не повлияет на другие Java-программы, а во-вторых, заметно упрощает поиск ошибок, связанных с некорректным значением classpath.

Наконец, возможно применять и альтернативные подходы к хранению пакетов и файлов с исходным и бинарным кодом. Например, в качестве такого хранилища может быть использована база данных. Более того, существует ограничение на размещение объявлений классов в .java-файлах, которое рассматривается ниже, а при использовании БД любые ограничения можно снять. Тем не менее, при таком подходе рекомендуется предоставлять утилиты импорта/экспорта с учетом ограничения для переходов из/в файлы.

3.3. Модуль компиляции

Модуль компиляции (complication unit) - хранится в текстовом .java-файле и является единичной порцией входных данных для компилятора. Он состоит из трех частей:

- объявление пакета;
- import-выражения;
- объявления верхнего уровня.

Объявление пакета одновременно указывает, какому пакету будут принадлежать все объявляемые ниже типы. Это выражение может отсутствовать, что означает, что эти классы располагаются в безымянном пакете (другое название - пакет по умолчанию).

import-выражения позволяют обращаться к типам из других пакетов по их простым именам, "импортировать" их. Эти выражения также необязательны.

Наконец, объявления верхнего уровня содержат объявления одного или нескольких типов. Название "верхнего уровня" противопоставляет эти классы и интерфейсы, располагающиеся в пакетах, внутренним типам, которые являются элементами и располагаются внутри других типов. Как ни странно, но эта часть также является необязательной, в том смысле, что компилятор не выдаст ошибки в случае ее отсутствия. Однако, никаких .class-файлов сгенерировано тоже не будет.

Доступность модулей компиляции определяется платформенной поддержкой, т.к. утилиты Java являются обычными программами, которые исполняются операционной системой по общим правилам.

Рассмотрим все 3 части более подробно.

3.3.1. Объявление пакета

Первое выражение в модуле компиляции - объявление пакета. Оно записывается с помощью ключевого слова package, после которого указывается полное имя пакета.

Например, первой строкой (после комментариев) в файле java/lang/Object.java идет:

```
package java.lang;
```

что служит одновременно объявлением пакета lang, вложенного в пакет java, и указанием, что объявляемый ниже класс Object, находится в этом пакете. Так складывается полное имя класса java.lang.Object.

Если это выражение отсутствует, то такой модуль компиляции принадлежит безымянному пакету. Этот пакет по умолчанию обязательно должен поддерживаться реализацией Java-платформы. Обратите внимание, что он не может иметь вложенных пакетов, так как составное имя пакета должно обязательно начинаться с имени пакета верхнего уровня.

Таким образом, самая простая программа может выглядеть следующим образом:

```
class Simple {
   public static void main(String s[]) {
      System.out.println("Hello!");
   }
}
```

Этот модуль компиляции будет принадлежать безымянному пакету.

Пакет по умолчанию был введен в Java для облегчения написания очень небольших или временных приложений, для экспериментов. Если же программа будет распространяться для пользователей, то рекомендуется расположить ее в пакете, который в свою очередь должен быть правильно назван. Соглашения по именованию рассматриваются ниже.

Доступность пакета определяется по доступности модулей компиляции, в которых он объявляется. Точнее, пакет доступен тогда и только тогда, когда выполняется любое из следующих двух условий:

- доступен модуль компиляции с объявлением этого пакета;
- доступен один из вложенных пакетов этого пакета.

Таким образом, для следующего кода:

```
package space.star;

class Sun {
}
```

если файл, который хранит этот модуль компиляции, доступен Java-платформе, то пакеты space и вложенный в него star (полное название space.star) также становятся доступны для Java.

Если пакет доступен, то область видимости его объявления - все доступные модули компиляции. Проще говоря, все существующие пакеты доступны для всех классов, никаких ограничений на доступ к пакетам в Java нет.

Требуется, чтобы пакеты java.lang и java.io, а значит и java, всегда были доступны для Java-платформы, поскольку они содержат классы, необходимые для работы любого приложения.

3.3.2. Импорт-выражения

Как будет рассмотрено ниже, область видимости объявления типа - пакет, в котором он располагается. Это означает, что внутри этого пакета допускается обращение к типу по его простому имени. Из всех других пакетов необходимо обращаться по составному имени, то есть полное имя пакета плюс простое имя типа, разделенные точкой. Поскольку пакеты могут иметь довольно длинные имена (например, дополнительный пакет в составе JDK1.2 называется com.sun.image.codec.jpeg), а тип может многократно использоваться в модуле компиляции, то такое ограничение может привести к усложнению исходного кода и сложностям в разработке.

Для решения этой проблемы вводятся import-выражения, позволяющие импортировать типы в модуль компиляции и далее обращаться к ним по простым именам. Существует два вида таких выражений:

- импорт одного типа;
- импорт пакета.

Важно подчеркнуть, что импортирующие выражения являются, по сути, подсказкой компилятора. Он пользуется ими, чтобы для каждого простого имени типа из другого пакета получить его полное имя, которое и попадает в компилированный код. Это означает, что импортирующих выражений может быть очень много, включая и такие, что импортируют неиспользуемые пакеты и типы,- все это никак не скажется ни на размере, ни на качестве бинарного кода. Также нет никакой разницы, обращаться ли к типу по его полному имени, или включить его в импортирующее выражение и обращаться по простому имени - результат будет идентичный.

Импортирующие выражения имеют эффект только внутри модуля компиляции, в котором они объявлены. Все объявления типов высшего уровня, находящиеся в этом же модуле, могут одинаково пользоваться импортированными типами. К импортированным типам допускается и обычный доступ по полному имени.

Выражение, импортирующее один тип, записываются с помощью ключевого слова import и полного имени типа. Например:

```
import java.net.URL;
```

Такое выражение означает, что в дальнейшем в этом модуле компиляции простое имя URL будет обозначать одноименный класс из пакета java.net. Попытка импортировать тип, недоступный на момент компиляции, вызовет ошибку. Если один и тот же тип импортируется несколько раз, то это не создает ошибки, а дублирующиеся выражения игнорируются. Если же импортируются типы с одинаковыми простыми именами из разных пакетов, то такая ситуация породит ошибку компиляции.

Выражение, импортирующее пакет, включает в себя полное имя пакета следующим образом.

```
import java.awt.*;
```

Это выражение делает доступными все типы, находящиеся в пакете java.awt, по их простому имени. Попытка импортировать пакет, недоступный на момент компиляции, вызовет ошибку. Импортирование одного пакета многократно не создает ошибки, дублирующиеся выражения игнорируются. Обратите внимание, что импортировать вложенный пакет нельзя.

Например:

```
// пример вызовет ошибку компиляции
import java.awt.image;
```

Можно предположить, что теперь возможно обращаться к типам пакета java.awt.image по упрощенному имени, например, image.ImageFilter. На самом деле пример вызовет ошибку компиляции, так как такое выражение расценивается как импорт типа, а в пакете java.awt отсутствует тип image.

Аналогично, выражение

```
import java.awt.*;
```

не делает доступнее классы пакета java.awt.image, их необходимо импортировать отдельно.

Поскольку пакет java.lang содержит типы, без которых невозможно создать ни одну программу, то он импортируется неявным образом в каждый модуль компиляции. Таким образом, все типы из этого пакета доступны по их простым именам без каких-либо дополнительных усилий. Попытка импортировать этот пакет еще раз будет проигнорирована.

Допускается одновременно импортировать пакет и какой-нибудь тип из него:

```
import java.awt.*;
import java.awt.Point;
```

Может появиться вопрос, как же лучше поступать - импортировать типы по отдельности, или сразу весь пакет? Есть ли какая-нибудь разница в этих подходах?

Разница заключается в алгоритме работы компилятора, который приводит каждое простое имя к полному. Он состоит из трех шагов:

- сначала просматриваются выражения, импортирующие типы;
- затем другие типы, объявленные в текущем пакете, в том числе, и в текущем модуле компиляции;
- наконец, просматриваются выражения, импортирующие пакеты.

Таким образом, если явно импортирован тип, то невозможно ни объявление нового типа с таким же именем, ни доступ по простому имени к одноименному типу в текущем пакете.

Например:

```
// пример вызовет ошибку компиляции
package my_geom;

import java.awt.Point;

class Point {
}
```

Этот модуль вызовет ошибку компиляции, так как имя Point в объявлении высшего типа будет рассматриваться как обращение к импортированному классу java.awt.Point, а его переопределять, конечно, нельзя.

Если в пакете объявлен тип:

```
package my_geom;

class Point {
}
```

то в другом модуле компиляции:

```
package my_geom;

import java.awt.Point;

class Line {
   void main() {
      System.out.println(new Point());
   }
}
```

складывается неопределенная ситуация - какой из классов, my_geom.Point или java.awt.Point, будет использован при создании объекта? Результатом будет:

```
java.awt.Point[x=0,y=0]
```

Как и регламентируется правилами, имя Point было трактовано на основе импорта типа. К классу текущего пакета все еще можно обращаться по полному имени: my_geom.Point. Если бы рассматривался безымянный пакет, то обратиться к такому "перекрытому" типу было бы уже невозможно, что является дополнительным аргументом к рекомендации располагать серьезные программы в именованных пакетах.

Теперь рассмотрим импорт пакета. Его еще называют "импорт по требованию", подразумевая, что не происходит никакой "загрузки" всех типов импортированного пакета сразу при указании импортирующего выражения, их полные имена подставляются по мере использования простых имен в коде. Возможно импортировать пакет и использовать только один тип (или даже ни одного) из него.

Изменим рассматриваемый выше пример:

```
package my_geom;

import java.awt.*;

class Line {
   void main() {
      System.out.println(new Point());
      System.out.println(new Rectangle());
   }
}
```

Теперь результатом будет:

```
my_geom.Point@92d342
java.awt.Rectangle[x=0,y=0,width=0,height=0]
```

Тип Point нашелся в текущем пакете, поэтому компилятору не пришлось делать поиск по пакету java.awt. Второй объект порождается от класса Rectangle, который не существует в текущем пакете, зато обнаруживается в java.awt.

Также корректен теперь пример:

```
package my_geom;

import java.awt.*;

class Point {
}
```

Таким образом, импорт пакета не препятствует объявлению новых или обращению к существующим типам текущего пакета по простым именам. Если все же нужно работать именно с внешними типами, то можно воспользоваться импортом типа или обращаться к ним по полным именам. Кроме этого, считается, что импорт конкретных типов помогает при прочтении кода сразу понять, какие внешние классы и интерфейсы используются в этом модуле компиляции. Однако полностью полагаться на такое соображение не стоит, так как возможны случаи, когда импортированные типы не используются и, напротив, в коде стоит обращение к другим типам по полному имени.

3.3.3. Объявление верхнего уровня

Далее модуль компиляции может содержать одно или несколько объявлений классов и интерфейсов. Подробно формат такого объявления рассматривается в следующих лекциях, однако приведем краткую информацию и здесь.

Объявление класса начинается с ключевого слова class, интерфейса - interface. Далее указывается имя типа, а затем в фигурных скобках описывается тело типа. Например:

```
package first;
```

```
class FirstClass {
}

interface MyInterface {
}
```

Область видимости типа - пакет, в котором он описан. Из других пакетов к типу можно обращаться либо по составному имени, либо с помощью импортирующих выражений.

Однако кроме области видимости в Java также есть средства разграничения доступа. По умолчанию тип объявляется доступным только для других типов своего пакета. Чтобы другие пакеты также могли использовать его, можно указать ключевое слово public:

```
package second;

public class OpenClass {
}

public interface PublicInterface {
}
```

Такие типы доступны для всех пакетов.

Объявления верхнего уровня описывают классы и интерфейсы, которые хранятся в пакетах. В версии Java 1.1 были введены внутренние (inner) типы, которые объявляются внутри других типов и являются их элементами наряду с полями и методами. Эта возможность является вспомогательной и довольно запутанной, поэтому не рассматривается подробно в курсе, хотя некоторые примеры и пояснения помогут в целом ее освоить.

Если пакеты, исходный и бинарный код хранятся в файловой системе, то Java может накладывать ограничение на объявления классов в модулях компиляции. Это ограничение создает ошибку компиляции в случае, если описание типа не обнаруживается в файле с названием, составленным из имени типа и расширения (например, ,java), и при условии:

- тип объявлен как public, и, значит, может быть использован из других пакетов;
- либо если тип используется из других модулей компиляции в своем пакете.

Эти условия означают, что в модуле компиляции может быть максимум один тип, отвечающий этим условиям.

Другими словами, в модуле компиляции может быть максимум один public тип, и его имя и имя файла должны совпадать. Если же в нем есть не-public типы, имена которых не совпадают с именем файла, то они должны использоваться только внутри этого модуля компиляции.

Если же для хранения пакетов используется БД, то такое ограничение не должно накладываться.

На практике же программисты зачастую помещают в один модуль компиляции ровно один тип, независимо от того, public он или нет. Это существенно упрощает работу с ними. Например, описание класса space.sun.Size хранится в файле space\sun\Size.java, а

бинарный код - в файле Size.class в той же директории. Именно так устроены все стандартные библиотеки Java.

Обращаем внимание, что при объявлении классов, вполне допускаются перекрестные обращения. Например, следующий пример совершенно корректен:

```
package test;

/*
 * Класс Human, описывающий человека
 */
class Human {
 String name;
 Car car; // принадлежащая человеку машина
}

/*
 * Класс Car, описывающий автомобиль
 */
class Car {
 String model;
 Human driver; // водитель, управляющий машиной
}
```

Кроме того, класс Car был использован ранее, чем был объявлен. Такое перекрестное использование типов также допускается в случае, если они находятся в разных пакетах. Компилятор должен поддерживать возможность транслировать их одновременно.

3.4. Уникальность имен пакетов

Поскольку Java создавалась как язык, предназначенный для распространения приложений через Интернет, а приложения состоят из структуры пакетов, то необходимо предпринять некоторые усилия, чтобы не произошел конфликт имен. Имена двух используемых пакетов могут совпасть по прошествии значительного времени после их создания. Исправить такое положение обычному программисту будет крайне затруднительно.

Поэтому создатели Java предлагают следующий способ уникального именования пакетов. Если программа создается разработчиком, у которого есть интернет-сайт, либо же он работает на организацию, у которой есть сайт, и доменное имя такого сайта, например, company.com, то имена пакетов должны начинаться с этих же слов, выписанных в обратном порядке: com.company. Дальнейшие вложенные пакеты могут носить названия подразделений компании, пакетов, фамилий, имена компьютеров и т.д.

Таким образом, пакет верхнего уровня всегда записывается ASCII-буквами в нижнем регистре и может иметь одно из следующих имен:

- трехбуквенные com, edu, gov, mil, net, org (этот список расширяется);
- двухбуквенные, обозначающие имена стран, такие как ru, su, de, uk и другие.

Если имя сайта противоречит требованиям к идентификаторам Java, то можно предпринять следующие шаги:

- если в имени стоит запрещенный символ, например тире, то его можно заменить знаком подчеркивания;
- если имя совпадает с зарезервированным словом, то можно в конце добавить знак подчеркивания;
- если имя начинается с цифры, можно в начале добавить знак подчеркивания.

Примеры имен пакетов, составленных по таким правилам:

```
com.sun.image.codec.jpeg
org.omg.CORBA.ORBPackage
oracle.jdbc.driver.OracleDriver
```

Однако, конечно, никто не требует, чтобы Java-пакеты были обязательно доступны на интернет-сайте, который дал им имя. Скорее была сделана попытка воспользоваться существующей системой имен вместо того, чтобы создавать новую для именования библиотек.

4. Область видимости имен

Областью видимости объявления некоторого элемента языка называется часть программы, откуда допускается обращение к этому элементу по простому имени.

При рассмотрении каждого элемента языка будет указываться его область видимости, однако полезно собрать в одном месте эту информацию.

Область видимости доступного пакета - вся программа, то есть любой класс может использовать доступный пакет. Однако необходимо помнить, что обращаться к пакету можно только по его полному составному имени. К п¿акету java.lang ни из какого места нельзя обратиться как к просто lang.

Областью видимости импортированного типа являются все объявления верхнего уровня в этом модуле компиляции.

Областью видимости типа (класса или интерфейса) верхнего уровня является пакет, в котором он объявлен. Из других пакетов доступ возможен либо по составному имени, либо с помощью импортирующего выражения, которое помогает компилятору воссоздать составное имя.

Область видимости элементов классов или интерфейсов - это все тело типа, в котором они объявлены. Если обращение к этим элементам происходит из другого типа, то тогда необходимо воспользоваться составным именем. Имя может быть составлено из простого или составного имени типа, имени объектной переменной или ключевых слов super или this, после чего через точку указывается простое имя элемента.

Аргументы метода, конструктора или обработчика ошибок видны только внутри этих конструкций и не могут быть доступы извне.

Область видимости локальных переменных начинается с момента их инициализации и до конца блока, в котором они объявлены. В отличие от полей типов, локальные переменные не имеют значений по умолчанию и должны инициализироваться явно.

```
int x;
for (int i=0; i<10; i++) {
   int t=5+i;
}
// здесь переменная t уже не доступна,
// так как блок, в котором она была объявлена
// уже завершен

// а переменная x еще не доступна,
// так как еще не была инициализирована
```

Определенные проблемы возникают, когда происходит перекрытие областей видимости и возникает конфликт имен различных конструкций языка.

4.1. "Затеняющее" объявление (Shadowing)

В классе Human (человек) объявлено поле age (возраст). Удобно определить также метод setAge(), который должен устанавливать новое значение возраста для человека. Вполне логично сделать у метода setAge() один входной аргумент, который также будет называть age (ведь в качестве этого аргумента будет передаваться новое значение возраста). Получается, что в реализации метода setAge() нужно написать age=age, в первом случае подразумевая поле класса, во втором - параметр метода. Понятно, что, хотя с точки зрения компилятора это корректная конструкция, попытка сослаться на две совершенно разные переменные через одно имя успехом завершиться не может. Надо заметить, что такие ошибки случаются порой даже у опытных разработчиков.

Во-первых, рассмотрим, из-за чего возникла конфликтная ситуация. Есть два элемента языка - аргумент метода и поле класса, области видимости которых пересеклись. Область видимости поля класса больше, она охватывает все тело класса. В то время как область видимости аргумента метода включает только сам метод. В таком случае внутри области пересечения по простому имени доступен именно аргумент метода, а поле класса "затеняется" (shadowing) объявлением параметра метода.

Остается вопрос, как все же обратиться к полю класса в такой ситуации. Если доступ по простому имени невозможен, надо воспользоваться составным. Здесь удобнее всего применить специальное ключевое слово this (оно будет подробно рассматриваться в дальнейших главах). Слово this имеет значение ссылки на объект, внутри которого оно применяется. Если вызвать метод setAge() у объекта класса Human и использовать слово this в этом методе, то его значение будет ссылкой на этот объект.

Исправленный вариант примера:

```
class Human {
   int age;// возраст

   void setAge(int age) {
      this.age=age; // верное присвоение!
   }
}
```

Конфликт имен, возникающий из-за затеняющего объявления, довольно легко исправить с помощью ключевого слова this или других конструкций языка в зависимости от обстоятельств. Наибольшей проблемой является то, что компилятор никак не сообщает о таких ситуациях, и самое сложное - выявить ее тестированием или контрольным просмотром кода.

4.2. "Заслоняющее" объявление (Obscuring)

Может возникнуть ситуация, когда простое имя может быть одновременно рассмотрено как имя переменной, типа или пакета.

Приведем пример, который частично иллюстрирует такой случай:

```
import java.awt.*;

public class Obscuring  {
   static Point Test = new Point(3,2);

   public static void main (String s[]) {
      print(Test.x);
   }
}

class Test {
   static int x = -5;
}
```

В методе main() простое имя Test одновременно обозначает имя поля класса Oscruing и имя другого типа, находящегося в том же пакете - Test. С помощью этого имени идет обращение к полю x, которое определено и в классе java.awt.Point и Test.

Результатом этого примера станет 3, то есть переменная имеет более высокий приоритет. В свою очередь, тип имеет более высокий приоритет, чем пакет. Таким образом, обращение к доступному в обычных условиях типу или пакету может оказаться невозможным, если есть более высокоприоритетное объявление одноименной переменной или типа. Такое объявление называется "заслоняющим" (obscuring).

Эта проблема скорее всего не возникнет, если следовать соглашениям по именованию элементов языка Java.

5. Соглашения по именованию

Для того чтобы код, написанный на Java, было легко читать и понимать не только его автору, но и другим разработчикам, а также для устранения некоторых конфликтов имен, предлагаются следующие соглашения по именованию элементов языка Java. Стандартные библиотеки и классы Java также следуют им, где это возможно.

Соглашения регулируют именование следующих конструкций:

- пакеты;

- типы (классы и интерфейсы);
- методы;
- поля;
- поля-константы;
- локальные переменные и параметры методов и др.;

Рассмотрим их последовательно.

Правила построения имен пакетов уже подробно рассматривались в этой главе. Имя каждого пакета начинается с маленькой буквы и представляет собой, как правило, одно недлинное слово. Если требуется составить название из нескольких слов, можно воспользоваться знаком подчеркивания или начинать следующее слово с большой буквы. Имя пакета верхнего уровня обычно соответствует доменному имени первого уровня. Названия java и javax (Java eXtension) зарезервированы компанией Sun для стандартных пакетов Java.

При возникновении ситуации "заслоняющего" объявления (obscuring) можно изменить имя локальной переменной, что не повлечет за собой глобальных изменений в коде. Случай же конфликта с именем типа не должен возникать согласно правилам именования типов.

Имена типов начинаются с большой буквы и могут состоять из нескольких слов, каждое следующее слово также начинается с большой буквы. Конечно, надо стремится к тому, чтобы имена были описательными, "говорящими".

Имена классов, как правило, являются существительными:

```
Human
HighGreenOak
ArrayIndexOutOfBoundsException
```

(Последний пример - ошибка, возникающая при использовании индекса массива, который выходит за допустимые границы).

Аналогично задаются имена интерфейсов, хотя они не обязательно должны быть существительными. Часто используется английский суффикс able:

```
Runnable
Serializable
Cloneable
```

Проблема "заслоняющего" объявления (obscuring) для типов редко встречается, так как имена пакетов и локальных переменных (параметров) начинаются с маленькой буквы, а типов - с большой.

Имена методов должны быть глаголами и обозначать действия, которое совершает данный метод. Имя должно начинаться с маленькой буквы, но может состоять из нескольких слов, каждое следующее слово начинается с заглавной буквы. Существует ряд принятых названий для методов:

- если методы предназначены для чтения и изменения значения переменной, то их имена начинаются с get и set соответственно. Например, для переменной size это будут getSize() и setSize().
- метод, возвращающий длину, называется length(), например, в классе String.
- метод, который проверяет булевское условие, имеет имя начинающееся с is, например, isVisible() у компоненты графического пользовательского интерфейса.
- метод, который преобразует величину в формат F, называется toF(), например, метод toSting(), который приводит любой объект к строке.

Вообще, рекомендуется везде, где возможно, называть методы похожим образом, как в стандартных классах Java, для того, чтобы они были легко понятны всем разработчикам.

Поля класса имеют имена, записываемые в том же стиле, что и для методов - начинаются с маленькой буквы, могут состоять из нескольких слов, каждое следующее слово начинается с заглавной буквы. Имена должны быть существительными, например, поле name в классе Human или size в классе Planet.

Как для полей решается проблема "заслоняющего" объявления (obscuring) уже обсуждалось.

Поля могут быть константами, если в их объявлении стоит ключевое слово final. Их имена состоят из последовательности слов, сокращений, аббревиатур. Записываются они только большими буквами, слова разделяются знаками подчеркивания:

```
PI
MIN_VALUE
MAX_VALUE
```

Иногда константы образуют группу, тогда рекомендуется использовать одно или несколько одинаковых слов в начале имен:

```
COLOR_RED
COLOR_GREEN
COLOR_BLUE
```

Наконец, рассмотрим имена локальных переменных и параметров методов, конструкторов и обработчиков ошибок. Они, как правило, довольно короткие, но тем не менее должны быть осмыслены. Например, можно использовать аббревиатуру (имя cp для ссылки на экземпляр класса ColorPoint) или сокращение (buf для buffer).

Распространенные однобуквенные сокращения:

```
byte b;
char c;
int i,j,k;
long l;
float f;
double d;
Object o;
String s;
Exception e; // объект, представляющий ошибку в Java
```

Двух- и трехбуквенные имена не должны совпадать с принятыми доменными именами первого уровня интернет-сайтов.

6. Заключение

В этой главе был рассмотрен механизм именования элементов языка. Для того чтобы различные части большой системы не зависели друг от друга, вводится понятия область видимости имени, вне которой необходимо использовать не простое, а составное имя. Затем было изучено важно понятия элементов (members), которые могут быть у пакетов и ссылочных типов. Также рассматривается связь терминов идентификатор (из темы Лексика) и имя.

Затем были рассмотрены пакеты, которые используются в Java для создания физической и логической структуры классов, а также и для более точного разграничения области видимости. Пакет содержит в себе вложенные пакеты и типы (классы и интерфейсы). Вопрос о платформенной поддержке пакетов привел к рассмотрению модулей компиляции как текстовых файлов, так как именно в виде файлов и директорий, как правило, хранятся и распространяются Java-приложения. Тогда же впервые был рассмотрен вопрос разграничения доступа, так как доступ к модулям компиляции определяется именно платформенной поддержкой, а точнее – операционной системой.

Модуль компиляции состоит из трех основных частей – объявление пакета, импорт-выражения и объявления верхнего уровня. Важную роль играет безымянный пакет, или пакет по умолчанию, хотя он и не рекомендуется для применения при создании больших систем. Были рассмотрены детали применения двух видов импорт-выражений – импорт класса и импорт пакета. Наконец, было начато рассмотрение объявлений верхнего уровня (эта тема будет продолжена в главе, описывающей объявление классов). Пакеты, как и другие элементы языка, имеют определенные соглашения по именованию, призванные облегчить понимание кода и уменьшить возможность появления ошибок и двусмысленных ситуаций в программе.

Описание области видимости для различных элементов языка приводит к вопросу о возможных перекрытиях таких областей и, как следствие, конфликтов имен. Рассматриваются «затеняющие» и «заслоняющие» объявления. Для устранения или уменьшения возможности возникновения таких ситуаций описываются соглашения по именованию для всех элементов языка.

7. Контрольные вопросы

5-1. Какие элементы языка Java имеют имена? Какие из них должны быть объявлены?

a.) Следующие элементы языка имеют имена:

- пакеты;
- классы;
- интерфейсы;
- элементы (member) ссылочных типов:
  - поля;

- методы;
- внутренние классы и интерфейсы;

• аргументы:
- методов;
- конструкторов;
- обработчиков ошибок;

• локальные переменные.

Все они должны быть объявлены, так как именно при объявлении указывается имя.

5-2. Что из перечисленных ниже слов является простым именем, составным именем, идентификатором?

```
MyClass
MyClass.name
MyClass.name.toString()
MyClass.name.toString().hashCode()
```

a.) MyClass – простое имя. MyClass.name, MyClass.name.toString – составные имена. Все они и hashCode – идентификаторы.

5-3. Могут ли пакет и вложенный пакеты содержать одноименные классы?

a.) Да, поскольку эти классы будут иметь непересекающуюся область видимости.

5-4. Из какой директории необходимо запускать компилятор, чтобы скомпилировать программу, состоящую из одного класса test.first.Start, описание которого сохранено в файле c:\Java\programs\test\first\Start.java?

a.) Компилятор можно запускать из любой директории, главное – правильно указать путь к файлу. Например,

```
c:\Java>javac programs\test\first\Start.java
```

или

```
c:\Java\programs\test>javac first\Start.java
```

5-5. Из какой директории необходимо запускать интерпретатор Java для выполнения программы, описанной в предыдущем вопросе?

a.) JVM нужно запускать из той директории, где лежит пакет по умолчанию, т.е.

```
c:\Java\programs>java test.first.Start
```

5-6. Можно ли исполнить программу, описанную в предыдущих 2 вопросах, если запускать виртуальную машину из директории c:\ ?

a.) Можно, для этого необходимо включить в classpath необходимый класс.

5-7. Ниже приведено несколько вариантов записи модуля компиляции. Какие из них корректны, если предполагается описать класс Point из пакета test.demo, причем класс активно использует классы java.awt.Point и несколько классов из пакета java.net?

```
a)
package test.demo;
import java.awt.Point;
import java.net.*;

b)
import java.awt.*;
import java.net.*;
package test.demo;

c)
package test.demo;
import java.net.*;
import java.awt.*;

d)
package test.demo.*
import java.net.*;
import java.awt.*;
```

a.) Ответ a) не годится, так как явное импортирование класса java.awt.Point не позволит объявить класс Point, как этого требует условие задачи. Ответ b) не корректен, так как объявление пакета должно идти до импорт-выражений. Ответ d) не верен, так объявление пакета должно содержать только имя пакета (безо всяких звездочек) и оканчиваться знаком точка с запятой. Ответ c) верен.

5-8. Корректен ли объявленный ниже класс? Если нет, то как его можно исправить?

```
class Box {
   private int weight=0;

   public int getWeight() {
      return weight;
   }

   void setWieght(int weight) {
      weight=weigth;
   }
```

```
}
```

а.) Такое объявление корректно с точки зрения, однако в методе setWeight() произошло «затеняющее» объявление. Чтобы устранить конфликт имен между аргументом метода и полем класса, необходимо переименовать одно из них. Как правило, меняют имя именно у аргумента, так как это затронет код лишь одного метода.

5-9. Корректно ли следующее объявление с точки зрения формального выполнения соглашений по именованию?

```
public class flat{
   private int floor_number;
   private int r; // количество комнат

   public int rooms() {
      retun r;
   }

   public int GetFloorNumber() {
      return floor_number;
   }
}
```

а.) Допущен целый ряд нарушений соглашений:

- Класс назван с прописной буквы, должно быть Flat.
- Имя поля floor_number содержит два слова, разделенных знаком подчеркивания, должно быть floorNumber.
- Имя поля r состоит из одной буквы, а оно должно быть более говорящим, например, rooms или roomsNumber
- Имя метода rooms является существительным, а должно быть глаголом, например, getRoomsNumber.
- Имя метода GetFloorNumber начинается с заглавной буквы, должно быть getFloorNumber.

Однако необходимо помнить, что соглашения призваны облегчать написание кода. Если есть какие-либо причины (например, они противоречат давно устоявшимся правилам написания программ), которые напротив усложняют процесс разработки, не нужны следовать соглашениям слишком формально.