Конспект лекций по Java. Занятие 2

В.Фесюнов

1. Отступление

Тема данного занятия является вводной, но тем не менее очень важной. Здесь вводится в рассмотрение ряд принципов объектно-ориентированного подхода, которые будут более подробно раскрыты на последующих занятиях.

Материал данного занятия может остаться не до конца понятым с первого раза. В этом случае можно порекомендовать продолжить знакомство со следующими темами и вернуться к нему позже.

2. JAVA — объектно-ориентированный язык программирования

Java является объектно-ориентированным языком программирования. Если сравнивать в этом смысле Java и C++, то между ними есть существенные различия.

C++ тоже является объектно-ориентированным языком. Но он является расширением C, который не является объектно-ориентированным. Из этого, в частности следует, что на C++ можно писать программы как в объектно-ориентированном стиле, так и в обычном. Т.е. на C++ можно программировать, не зная и не понимая объектно-ориентированного подхода. Можно даже смешивать оба подхода в едином продукте или пытаться программировать в объектно-ориентированном стиле, а тогда, когда это не получается, скатываться к традиционному программированию.

В Java так нельзя. В ней нет средств, позволяющих писать не объектно-ориентированные программы.

Из этого сразу следует один вывод. *Нельзя научиться программировать на Java, не*

овладев основами объектно-ориентированного подхода.

2.1. 5 принципов объектно-ориентированного подхода

- Все является объектом
  Все данные программы хранятся в объектах. Каждый объект создается (есть средства для создания объектов), существует какое-то время, потом уничтожается.
- Программа есть группа объектов, общающихся друг с другом
  Кроме того, что объект хранит какие-то данные, он умеет выполнять различные операции над своими данными и возвращать результаты этих операций. Теоретически эти операции выполняются как реакция на получение некоторого сообщения данным объектом. Практически это происходит при вызове метода данного объекта. Что такое метод и как он относится к объекту, мы рассмотрим позднее.
- Каждый объект имеет свою память, состоящую из других объектов и/или элементарных данных.
  Объект хранит некоторые данные. Эти данные — это другие объекты, входящие в состав данного объекта и/или данные элементарных типов, такие как целое, вещественное, символ, и т.п.
- Каждый объект имеет свой тип (класс)
  Т.е. в объектно-ориентированном подходе не рассматривается возможность создания произвольного объекта, состоящего из того, например, что мы укажем в момент его создания. Все объекты строго типизированы. Мы должны сначала описать (создать) тип (класс) объекта, указав в этом описании из каких элементов (полей) будут состоять объекты данного типа. После этого мы можем создавать объекты этого типа. Все они будут состоять из одних и тех же элементов (полей).
- Все объекты одного и того же типа могут получать одни и те же сообщения
  Кроме описания структуры данных, входящих в объекты данного типа, описание типа содержит описание всех сообщений, которые могут получать объекты данного типа (всех методов данного класса). Более того, в описании типа мы должны задать не только перечень и сигнатуру сообщений данного типа, но и алгоритмы их обработки.

2.2. Реализация принципов объектно-ориентированного

подхода в Java

2.2.1. Ссылки на объекты

В Java для манипулирования объектами в программном коде используются *ссылки на объекты (handles)*. Ссылка хранит в себе некоторый адрес объекта в оперативной памяти.

Может быть несколько ссылок на один объект. На какой-то объект может вообще не быть ссылок (тогда он для нас безвозвратно потерян). Ссылка может не ссылаться ни на какой объект — пустая (**null**) ссылка. Не может быть ссылки в никуда или ссылки на какую-то произвольную область памяти. Как транслятор Java, так и JVM внимательно следят за тем, чтобы нельзя было создать ссылку на какую-то произвольную область памяти. Практически в Java это сделать невозможно.

2.2.2. Все ссылки имеют имя

Для манипулирования самими ссылками в программном коде необходимо как-то их обозначать. Это делается при помощи имени ссылки. Все ссылки, так или иначе, описываются, при этом каждой ссылке дается имя. Имена ссылок известны программе и встречаются в программном коде там, где нужно манипулировать объектами, на которые они ссылаются.

2.2.3. Все ссылки строго типизированы

При описании ссылки обязательно указывается ее тип. И эта ссылка может ссылаться только на объект данного типа. Есть определенные исключения из этого правила, связанные с наследованием классов. Мы рассмотрим это позднее.

Попытка присвоить ссылке адрес объекта не того класса пресекается как на этапе трансляции программы (выдаются ошибки трансляции), так и на этапе ее выполнения (возникает исключительная ситуация `ClassCastException`).

Приведем пример описания ссылки

```
    MyType ref;
```

Здесь `MyType` — имя типа (как и ссылки, все типы имеют имя), `ref` — имя ссылки. После такого описания ссылке `ref` можно присвоить значение — адрес какого-то объекта типа `MyType`.

2.2.4. Создание объектов

Все объекты в Java создаются только явно, для чего используется операция **new**.

```
ref = new MyType();
```

Здесь создается объект типа `MyType` и его адрес заносится в `ref`. Почему здесь использованы скобки после `MyType`, мы рассмотрим позже. Еще один пример

```
MyType ref = new MyType();
```

Здесь описание ссылки совмещено с инициализацией.

2.2.5. Класс — способ описания типа

Для описания типов в Java используется механизм **классов**. За исключением базовых (иначе — элементарных) типов (`int, char, float` и др.) и интерфейсов (что это такое, мы рассмотрим позже), все остальные типы — это классы.

В простейшем случае описание класса выглядит так

```
class MyClass {

. . . // тело класса

}
```

Здесь **class** — ключевое слово, `MyClass` — имя класса. Внутри фигурных скобок находится тело класса.

Внутри тела класса описываются в произвольном порядке поля и методы класса.

Отступление

- Существуют общепринятые правила именования классов, их полей и методов. Следование этим правилам улучшает "читабельность" программы. Имена классов принято начинать с большой буквы, а имена полей и методов — с маленькой. Если имя состоит из нескольких слов, то каждое новое слово начинают с большой буквы.
- Общепринятых правил относительного размещения описаний полей и методов не существует. Но лучше, все же, размещать вместе все описания полей (например, в начале или в конце описания класса), а после или перед ними — описания методов.
- В примере использован комментарий. В Java два типа комментариев. Все, что начинается с двух символов '/', является комментарием и этот комментарий продолжается до конца данной строки. Все, что начинается с символов "/*" является комментарием, который должен быть закрыт символами "*/".

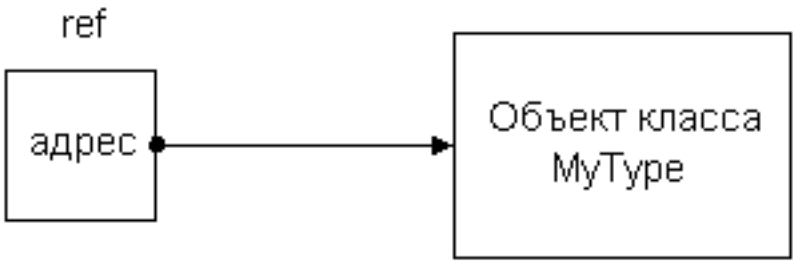
После того, как класс описан, мы можем создавать *объекты* (*objects*) класса (иначе — *экземпляры класса, class instances*).

2.2.6. Данные элементарных типов ссылками не являются

Ссылка хранит адрес объекта, а объект уже хранит какую-то содержательную информацию. В отличие от ссылок, данные элементарных типов являются самосодержащими, они сами хранят содержательную информацию.

Так можно продемонстрировать ссылку на объект.

```
MyType ref = new MyType();
```



А так — данное элементарного типа.

int var = 3;

Тип	Описатель	Размер	Комментарий
Логический	boolean	?*	-
Символьный	char	2 байта	Unicode
Байтовый	byte*	1 байт	(-128 - 127)
Короткий целый	short	2 байта	(-215 — 215-1)
Целый	int	4 байта	(-231 — 231-1)
Длинный целый	long	8 байт	(-263 — 263-1)
Вещественный	float	4 байта	-
Вещественный двойной точности	double	8 байт	-
Пустой	void*	-	-

Таблица 1. Базовые типы Java.

Замечание:

В спецификации языка размер типа `boolean` явно не специфицирован. Указано лишь, что он может принимать всего два значения — `true` или `false` (для этого достаточно одного бита). Но в спецификации Java-машины (JVM) указано, что для представления переменной или поля типа `boolean` используется тип `int`, а для массива `boolean` используется массив `byte`.
Тип `byte` является арифметическим целым типом.
Все целые типы в Java знаковые, соответствующих беззнаковых эквивалентов, как в C++, в Java нет.
Тип `void` фактически типом не является. Описатель `void` просто может быть использован при описании метода для указания того, что этот метод ничего не возвращает.

2.2.7. Поля класса и переменные программы

Как уже отмечалось, в классе можно описать *поля класса* (*fields or instance variable*). Поля класса определяют, из каких данных будут состоять объекты этого класса. Поля могут быть ссылками на другие объекты или элементарными данными.

В методах класса могут быть описаны *переменные*. Их не следует путать с полями класса. Как и поле класса, переменная может быть либо ссылкой, либо данным базового типа. Описание переменной выглядит точно так же, как и описание поля класса, за исключением того, что ряд описателей не применимы для переменных. Отличаются же (визуально) переменные от полей местом их описания. Поля класса описываются непосредственно в теле класса, на том же уровне вложенности, что и методы класса. Переменные описываются внутри методов. Пример:

```
class SomeClass {  // Это заголовок класса

    int i = 0;              // Это элементарное данное, поле класса
    MyType ref;      // Это ссылка, тоже поле класса

    int f() {                // Это заголовок метода
      int k = 0;                // Это элементарное данное, переменная
      MyType lref;              // Это ссылка, переменная

      . . .                     // Данный метод что-то делает

    }                           // Это конец метода

    . . .                     // В классе могут быть и другие методы

}                             // Это конец тела класса
```

2.2.8. Область видимости и время жизни переменных

В различных языках программирования существуют различные типы или классы переменных — локальные, глобальные, статические и т.п. В Java только один тип переменных — локальные переменные. *Время жизни* переменной в Java определяется правилом:

- Переменная создается в точке ее описания и существует до момента окончания того блока, в котором находится данное описание.

В Java *блок* — это то, что начинается открывающей фигурной скобкой '{' и заканчивается закрывающей фигурной скобкой '}'.

Областью видимости переменной (*scope*) является фрагмент программы от точки ее описания до конца текущего блока.

- Область видимости — это статическое понятие, имеющее отношение к какому-то фрагменту текста программы. Время жизни, в отличие от области видимости, — это понятие динамики выполнения программы. Время жизни переменных в Java совпадает с их областью видимости с учетом отличия самих этих понятий.

Если в блоке, где описана данная переменная, вложены другие блоки, то переменная доступна в этих блоках (обычная практика языков программирования). Но, в отличие от многих других языков, в Java запрещено переопределять переменную во вложенных блоках (т.е. описывать другую переменную с тем же именем).

Рассмотрим примеры, демонстрирующие эти понятия.

Пример 1

```
. . .   // предшествующая часть программы

{       // начало блока

    . . .  // какие-то операторы внутри блока

    int x = 1;   // 1-я точка описания

    {            // еще один блок

        . . .    // какие-то операторы внутри блока

        int y;   // 2-я точка описания

        . . .    // какие-то операторы внутри блока

    }  // переменная y уничтожается и она больше не видна

    . . .  // какие-то операторы внутри блока
```

```
}  // переменная x уничтожается и она больше не видна

. . .   // последующая часть программы
```

Пример 2

```
{       // начало блока

    int x = 1;
    long y;

    {           // еще один блок

        int x;   // !!! ошибка
        int y;   // !!! ошибка

        . . .    // какие-то операторы внутри блока

    }

    . . .  // какие-то операторы внутри блока

}
```

Смысл примеров указан в комментариях в самих примерах. Первый пример просто показывает, в каких пределах текста видна та или иная переменная (область видимости) и, когда она создается и уничтожается (время жизни). Второй пример содержит ошибки. Он демонстрирует запрет переопределять переменные во вложенных блоках.

2.2.9. Область видимости и время жизни объектов

Иная картина наблюдается с объектами. Объекты доступны в программе только через ссылки на них. Поэтому область видимости объекта определяется областью видимости ссылок на этот объект (на один объект может быть сколько угодно ссылок).

Время жизни объекта определяется следующим правилом.

- Объект существует, пока существует хотя бы одна ссылка на этот объект.

Это правило, однако, не утверждает, что объект будет уничтожен, как только пропадет

последняя ссылка на него. Просто такой объект становится недоступным и может быть уничтожен.

- В Java нет явного уничтожения объектов. Объекты уничтожаются (говорят — утилизируются) *сборщиком мусора* (*garbage collector*), который работает в фоновом режиме параллельно с самой программой на Java.

Рассмотрим следующий фрагмент.

```
{
  SomeType localReference = new SomeType();
  globalReference = localReference;
}
```

Здесь `SomeType` — это некоторый класс, `localReference` — локальная переменная-ссылка, `globalReference` — некоторая внешняя, по отношению к данному блоку, переменная или поле класса (из данного фрагмента нельзя сделать однозначный вывод, что это).

В этом фрагменте порождается объект класса `SomeType` и адрес этого объекта заносится в переменную `localReference`. После этого этот же адрес из `localReference` копируется в `globalReference`. По выходу из блока переменная `localReference` уничтожается, но переменная (или поле) `globalReference` продолжает существовать. Соответственно, продолжает существовать и порожденный объект.

2.2.10. Описание методов класса

В первом приближении *методы класса* (*class methods*) можно рассматривать как функции.

Описание метода выглядит следующим образом

```
<тип> <имя_метода> (<аргументы>) {
     <тело_метода>
}
```

Здесь <тип> — это один из базовых типов (см. таблицу выше) или пользовательский тип (т.е. некоторое имя класса). <аргументы> — это список, возможно пустой,

параметров метода. <тело_метода> — собственно программный код данного метода.

Каждый аргумент или параметр метода в данном описании — это пара "<тип> <имя_аргумента>". Аргументы отделяются друг от друга запятыми.

Описания методов расположены внутри класса, на том же уровне вложенности скобок, что и описание полей класса. Не может быть описания метода вне класса или внутри другого метода или блока.

2.2.11. Вызов методов

Вызов методов отличается от вызовов функций в не объектно-ориентированных языках программирования. При вызове обычного (не статического) метода класса обязательно должен быть указан объект этого класса и метод вызывается для этого объекта. Т.е. вызов метода — это вызов метода объекта.

Формальное исключение составляет вызов метода класса из другого (или того же) метода данного класса, в этом случае объект можно не указывать. Но фактически объект и в данном случае имеется, это — тот объект, для которого был вызван вызывающий метод.

Рассмотрим это на примерах. Опишем класс `SomeClass` и в нем методы `f` и `g`.

```
class SomeClass {

     int f(int k) {
        . . .
     }

     void g() {
         . . .
      }
}
```

Здесь описан метод `f` с одним параметром целого типа, возвращающий целое значение и метод `g` без параметров, не возвращающий никакого значения. Приведем примеры вызова этих методов из некоторого фрагмента программы.

```

```

```
a.f(x);
b.g();
v = b.f(3);
```

В приведенном фрагменте фигурируют переменные (или поля класса) `a, x, b` и `v`. Переменные `a` и `b` должны быть описаны как ссылки с типом `SomeClass`, переменные `x` и `v` должны быть целочисленными.

Данный фрагмент демонстрирует, что объект, для которого вызывается метод, должен быть указан при помощи ссылки, имя которой записывается перед именем метода через точку.

При вызове метода класса из метода того же класса объект указывать не обязательно. Это, как указано выше, не нарушает того правила, что при вызове метода всегда должен быть определен объект, для которого этот метод вызывается. Просто в данном случае этот объект уже определен при вызове "вызывающего" метода и для него же вызывается "вызываемый" метод.

2.2.12. Доступ к полям класса

Поля класса не существуют сами по себе (за исключением статических). Они расположены внутри объекта класса. Поэтому при доступе к полю должен быть определен объект.

Как и в случае вызова метода, при обращении к полю класса извне класса объект должен быть указан явно (при помощи ссылки на объект) перед именем поля через точку. Например, если в классе `SomeClass` есть поля `fld1` и `fld2`, а `obj` — ссылка на объект класса `SomeClass`, то

```
obj.fld1 = 2;
x = obj.fld2;
```

являются примерами доступа к полям `fld1`, `fld2`.

Изнутри класса, т.е. из нестатических методов класса, можно обращаться к полям класса напрямую, без указания объекта, поскольку такой объект определен при вызове данного метода.

2.2.13. Передача параметров

Для параметров функций, методов, процедур в программировании существует понятие тип передачи параметра. Например, существуют понятия "передача параметра по значению" и "передача параметра по ссылке". В Java существует всего один тип передачи параметров — передача по значению. Это означает, что при вызове метода ему передается текущее значение параметра. Внутри метода можно произвольно изменять параметр, но это никак не повлияет, скажем, на переменную, которая была указана в качестве параметра вызова. Дело в том, что при передаче параметра выделяется необходимая область памяти, куда копируется значение параметра, и внутри метода работа идет с этой копией. Она будет уничтожена при выходе из метода.

Это нужно хорошо себе представлять, в особенности, когда передаются ссылки на объекты.

Рассмотрим пример. Пусть `ref` — ссылка на объект, передаваемая в качестве параметра при вызове некоторого метода `h(...)`.

```
h(ref);
```

Внутри метода `h` мы можем изменить параметр метода (т.е. присвоить ему ссылку на другой объект), но это никак не повлияет на саму ссылку `ref`, т.к. при вызове создается копия `ref` и изменяется именно она. С другой стороны мы можем внутри `h` менять данные того объекта, на который ссылается `ref`, и это реально отразится на этом объекте, т.к. создается копия только ссылки, но не самого объекта.