OPEN SOURCE GITHUB BOOK

JAVA

МАНУАЛ К ЛАБОРАТОРНЫМ РАБОТАМ
И НЕ ТОЛЬКО

ARTHUR KUPRIYANOV

Обратного пути нет. Все мы по началу думаем, что сможем завязать с программированием...

Письмак

Дисклеймер

Данная статья **не является официальной** со стороны университета ИТМО или кафедры ВТ

В мануале с малой долей вероятности могут быть ошибки или неточности. Сообщить о них можно по VK : https://vk.com/apploidxxx

Этот мануал не сборник ответов к вопросам, которые зададут вам практики на первой лабе. Он скорее поможет вам достичь **цели** первой лабораторной работы - "понять с чем вы имеете дело".

Примеры приведенных кодов в каталоге в гитхаб репозитории, там же вы можете найти раздельные файлы по главам в виде markdown разметки: https://github.com/AppLoidx/programming-manual

О Java в целом

Язык Java. Особенности языка.

Меня всегда забавлял этот контрольный вопрос в описании первой лабораторной работы, потому что он слишком абстрактный и непонятно, что именно нужно знать и отвечать на этот вопрос.

Так в чем же особенность языка Java? Думаю, самая всем известная его особенность - это кроссплатформенность.

> Ну, кроссплатформенность это, конечно, хорошо, но каким образом она достигается?

Чтобы понять это, давайте сначала рассмотрим прародитель языка Java, всеми известный C++. Вашу программу, написанную на языке Си, нужно будет компилировать под разные целевые платформы (Windows, Mac и прочее), которые будут работать только под них.

> А что значит компиляция? Почему под разные платформы вам нужно создать разные скомпилированные программы?

Опять появляется множество вопросов, на которые ответить, не зная основ компьютерной архитектуры - не так уж и просто ответить.

Начнем с архитектуры вашего любимого устройства:

Более подробную информацию вы можете найти здесь или прочитав книгу "Архитектура компьютера" Э.Таненбаума (я бы не советовал сейчас нагружать ваш мозг этим, но рано или поздно вам все равно, скорее всего, придется прочитать его (ОПД))

А здесь я попытаюсь рассказать вкратце и поверхностно.

Существует огромная разница между тем, что удобно людям, и тем что могут компьютеры. Мы хотим делать X, но компьютеры в то же время могут делать только Y. Из-за этого возникает проблема.

Эту проблему можно решить двумя способами. Оба подразумевают разработку новых команд, более удобных для человека, чем встроенные машинные команды. Эти новые команды в совокупности формируют язык, который будем называть Я1. Встроенные машинные команды - Я0. Компьютер может исполнять только программы, написанные на его машинном языке Я0.

Логично предположить, что в любом случае нам нужно **исполнить программу написанную на Я1, когда компьютеру доступен лишь язык Я0**.

Первый способ исполнения программы, написанной на языке Я1, подразумевает замену каждой команды эквивалентным набором команд на языке Я0. В этом случае компьютер исполняет новую программу, написанную на языке Я0, вместо старой программы, написанной на Я1. Эта технология называется **трансляцией**.

Трансляторы, которые транслируют программы на уровень 3 или 4 (см. рисунок выше), называются **компиляторами**.

Есть также второй способ, который заключается в создании на языке Я0 программы, получающей в качестве входных данных программы, написанные на языке Я1. При этом каждая команда языка Я1 обрабатывается поочередно, после чего сразу исполняется эквивалентный набор команд языка Я0. Эта технология не требует составления новой программы на Я0. Она называется **интерпретацией**, а программа которая осуществляет интерпретацию, называется **интерпретатором**.

А теперь представим себе **виртуальную машину**, для которой машинным языком является язык Я1. Назовем такую машину М1, а машину для работы с языком Я0 - М0. Если бы такую машину М1 можно было сконструировать без больших денежных затрат, язык Я0 был бы не нужен. Можно было бы писать программы сразу на языке Я1, а компьютер сразу бы их исполнял. Тем не менее, такие машины, возможно, не удастся создать из-за чрезмерной дороговизны или трудностей разработки. Поэтому и появилось понятие виртуальная машина. Люди вполне могут писать ориентированные на неё программы. Эти программы будут транслироваться или интерпретироваться программой, написанной на языке Я0, а сама она могла бы исполняться существующим компьютером. Другими словами, можно писать программы для виртуальных машин так, будто эти машины реально существуют.

JVM - это виртуальная машина, но не стоит путать её с **System virtual machines** (которые могут обеспечивать функциональность, необходимую для выполнения целых операционных систем).

JVM относится к **Process virtual machines**, которые предназначены для выполнения компьютерных программ в независимой от платформы среде. Например, он исполняет **байт-код**, который можно считать языком Я1, а машина на которой стоит наш JVM (М1) это М0, умеющий выполнять программы Я0. Другими словами, JVM физически не существует - это по сути программа, написанная на языке Я0, которая может обрабатывать программы с языком Я1, интерпретируя его в язык Я0.

Таким образом, JVM разный под каждую платформу, так как ему нужно интерпретировать входную программу (байт-код) в программу, которую может понять конкретная платформа (Windows, Mac etc).

Давайте поймем разницу между байт-кодом (программа для JVM) и двоичным кодом, который понимает "процессор":

Вот пример машинного кода и его представления на языке Ассемблера. Слева указан порядковый номер (адрес) первого байта команды. Во второй колонке мы видим байты команды, они записаны в восьмеричной системе счисления. В третьей колонке мнемоники Ассемблера, которые упрощают восприятие программы человеком.

```
004: 003 010       lbl adda    #8      immediate value decimal
006: 103 010           addb    #010    same thing in octal
010: 024 001           lda     b       memory reference
012: 235 220           stx     (ptr)   indirect reference
014: 306 204           ora     data,x  indexed
016: 337 220           lnega   (ptr),x indirect/indexed
020:               # jumps and calls
020: 344 004           jmp     lbl     unconditional jump
022: 043 030           jane    lbl2-2  jump if a not equal 0
024: 257 221           jxgt    (ptr+one) jump indirect if x gt
026: 364 041           call    sub     call to subroutine
```

```
030: 174 220           cbeq    (ptr)   call indirect if b eq 0
032:                # set and skips
032:                    lbl2
032: 122 204           set1    fox,2  set bit 2 of data to 1
034: 272 205           skp0    data+1,7  skip if bit 7 is 0
036:                # shifts and rotates
036: 001               shra    1+SHIFT   shift a right 4 plcs
037: 361               rolb    2       rotate b left 2 places
040: 000               hlt             halt
041: 000               sub db 0            return address
042: 200               nop             no op
043: 023 222           lda #0222
045: 123 144           ldb #100
047: 360               sys             system call (extension)
050: 023 222           sysp 0222,100   system call using parms
052: 123 144
054: 360
055: 354 041           jmp (sub)       return
```

Не надо вдаваться в подробности, просто пример изнутри.

Теперь о байт-коде. Основной проблемой двоичного кода является его специфичность. Два разных устройства, например, ноутбук и мобильный телефон, имеют кардинально разные процессоры и кардинально разные наборы команд и кодов.

Один из способов проблемы переносимости и сложности это *промежуточная виртуальная машина*.

Виртуальный процессор работает также, как и реальный: он видит массив чисел, и воспринимает их как команды для выполнения. Байт-код внешне совершенно идентичен двоичному коду. Вот пример байт-кода виртуальной машины Java:

```
000: 03        iconst_0
001: 3b        istore_0
002: 84 00 01  iinc 0, 1
005: 1a        iload_0
006: 05        iconst_2
007: 68        imul
010: 3b        istore_0
011: a7 ff f9  goto -7
```

Единственная разница заключается в том, что двоичный код исполняет физический процессор, а байт-код — очень простая программа-интерпретатор.

Итак, сделаем заключение:

Байт-код - это промежуточное представление программы, не привязанное к конкретной машинной архитектуре. Независимость от архитектуры машины обеспечивает переносимость, означающую, что уже разработанное (или скомпилированное) программное обеспечение может работать на любой платформе, поддерживающей JVM и абстракции языка Java.

> В настоящее время язык программирования Java в значительной степени независим от виртуальной машины Java, так что буква "J" в аббревиатуре "JVM" немного вводит в заблуждение, поскольку JVM в состоянии выполнять любой язык JVM, который может сгенерировать корректный файл класса. Например, Scala, генерирующий байт-код для выполнения в JVM.

Как итог, можно сказать, что если вы напишите вашу программу в Windows (и допустим сделаете из него какой-нибудь jar-файл), то он сможет запуститься на Mac или Unix (по крайней мере, так задумано), если у них стоит JVM.

Практика

Давайте напишем простую программу *Hello.java*:

```
public class Hello {
        public static void main(String ... args){
                for(int i=0;i<10;i++){
                        System.out.println("ITMO");
                }
        }
}
```

Скомпилируем её с помощью команды `javac`:

```
javac Hello.java
```

После того как вы её скомпилируете вы можете увидеть файл Hello.class - ваша скомпилированная программа, иначе говоря байт-код.

Запустить её можно командой `java`:

```
java Hello
```

Прим. не надо указывать его расширение (.class) - необходимо и достаточно указать лишь его имя.

Java поставляется с дизассемблером файлов классов под названием `javap`, который позволяет изучать .class-файлы. Взяв файл класса Hello и запустив `javap -c Hello`, мы получим следующий результат:

```
Compiled from "Hello.java"
public class Hello {
  public Hello();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."
<init>":()V
       4: return

  public static void main(java.lang.String...);
    Code:
       0: iconst_0
       1: istore_1
       2: iload_1
```

```
       3: bipush        10
       5: if_icmpge     22
       8: getstatic     #2                  // Field
java/lang/System.out:Ljava/io/PrintStream;
      11: ldc           #3                  // String ITMO
      13: invokevirtual #4                  // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
      16: iinc          1, 1
      19: goto          2
      22: return
}
```

Опять же пока не стоит вдаваться в подробности (хотя в моей памяти, вроде и бывало что спрашивали про основные команды, например, `goto`, `bipush` или `istore`)

Осталось чуть-чуть...

Итак, переходя ко второму контрольному вопросу: что же такое JVM, JRE и JDK?

С JVM мы в принципе более-менее разобрались - это виртуальная машина, на которой выполняются байт-коды.

JRE и JDK относительно проще, чем определение JVM:

- JRE (**Java Runtime Environment**) - это среда выполнения Java - она, помимо прочего, содержит JVM и является тем, что вам нужно для запуска Java-программы. Она не содержит инструментов и утилит, таких как компиляторы или отладчики для разработки приложений.
- JDK (**Java Development Kit**) - является расширенным набором JRE и содержит все, что есть в JRE, а также такие инструменты, как компиляторы и отладчики, необходимые для разработки.

JIT-компиляция

Еще один из частых вопросов на лабах - что такое JIT-компиляция?

JIT (Just-in-Time, своевременная) компиляция появилась в HotSpot VM (см. приложение A), в котором модули программы (интерпретированные из байт-кода) компилируются в машинный код. Модулями компиляции в HotSpot являются метод и цикл.

JIT-компиляция работает путем мониторинга приложения, выполняемого в режиме интерпретации, и выявления наиболее часто выполняемых фрагментов кода. В ходе анализа собирается информация, которая позволяет выполнять более сложную оптимизацию. Когда выполнение некоторого конкретного метода переходит установленный порог, профайлер запрашивает компиляцию и оптимизацию этого фрагмента кода.

JIT-подход к компиляции имеет много преимуществ, но одним из главных является то, что он основан на данных трассировки, собранной на этапе интерпретации, что позволяет HotSpot принять более обоснованные и разумные решения, касающиеся оптимизации.

После трансляции исходного кода Java в байт-код и еще одного этапа JIT-компиляции фактически выполняемый код очень существенно отличается от написанного исходного кода. Это ключевой момент, который будет управлять нашим подходом к исследованиям производительности. Код после JIT-компиляции, выполняющийся виртуальной машиной, может выглядеть не имеющим ничего общего с оригинальным исходным кодом на Java.

Пример инлайнинга JIT-компилятора можно увидеть здесь: https://habr.com/ru/post/305894/

Переходя к программированию

Далее будут материалы касающееся непосредственно программирования на языке Java.

Примитивные типы данных в Java

Рассмотрим примитивные типы в JVM:

Виртуальная машина Java поддерживает следующие примитивные типы: числовые типы, boolean тип и типы с плавающей точкой

Целые типы

- **byte**, содержит 8-битовые знаковые целые.
    - Значение по умолчанию - ноль.
- **short**, содержит 16-битовые знаковые целые.
    - Значение по умолчанию - ноль.
- **int**, содержит 32-битовые знаковые целые.
    - Значение по умолчанию - ноль.
- **long**, содержит 64-битовые знаковые целые.
    - Значение по умолчанию - ноль.
- **char,** содержит 16-битовые беззнаковые целые, представляющие собой кодовые точки таблицы символов Unicode в базовой странице UTF-16.
    - Значение по умолчанию - нулевая кодовая точка ('\u0000')

Типы с плавающей точкой

- **float**, содержит числа с плавающей точкой одинарной точности.
    - Значение по умолчанию - положительный ноль.
- **double**, содержит числа с плавающей точкой двойной точности.
    - Значение по умолчанию - положительный ноль.

Значение boolean типа может быть `true` или `false`, значение по умолчанию `false`.

Целочисленные типы и их значения

Существуют следующие диапазоны для целочисленных значений:

- для типа **byte** от -128 до 127 (-2^7^ до 2^7^ - 1) включительно;
- для типа **short** от -32768 до 32767 (-2^15^ до 2^15^ - 1) включительно;
- для типа **int** от -2147483648 до 2147483647 (-2^31^ до 2^31^ - 1) включительно;
- для типа **long** от -9223372036854775808 до 9223372036854775807 (-2^63^ до 2^63^ - 1) включительно;
- для типа **char** от 0 до 65535 включительно;

Запоминать эти значения наизусть не надо, но можно хотя бы примерно представлять их границы по степеням двойки.

Например, может попасться задача такого рода:

```java
public class ExampleWithByte{
    public static void main(String ... args){
        byte x = 127;
        x++;
        System.out.println(x);      //  -128
    }
}
```

Эта программа при исполнении выводит -128. Объяснить это очень просто, зная диапазоны типов данных. А почему именно -128 это вопрос к дискретке и двоичному представлению чисел в машине.

Типы данных с плавающей точкой, множества значений и значения

Типами данных с плавающей точкой являются типы **float** и **double** соответственно 32-х битые значения одинарной точности и 64-х битные значения двойной точности. Формат чисел и операции над ними соответствуют спецификации *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std. 754-1985, New York).

Стандарт IEEE 754 включает в себя не только положительные и отрицательные значения мантиссы, но также и положительные и отрицательные нули, положительные и отрицательные *бесконечности*, и специальное не числовое значение **NaN** (Not-a-Number). NaN используется в качестве результата некоторых неверных операций, таких как деление нуля на нуль.

Все значения (кроме не-чисел NaN) множества чисел с плавающей точкой *упорядочены*. Если числа упорядочить по возрастанию, то они образуют такую последовательность: отрицательная бесконечность, отрицательные конечные значения, отрицательный ноль, положительный ноль, положительные значения и положительная бесконечность.

Сравнивая положительный и отрицательный ноль, мы получим верное равенство, однако существуют операции, в которых их можно отличить; например, деля 1.0 на 0.0, мы получим положительную бесконечность, но деля 1.0 на -0.0 мы получим отрицательную бесконечность.

Не-числа NaN *не упорядочены*, так что сравнение и проверка на равенство вернёт *ложь*, если хотя бы один из операндов не-число NaN. В частности проверка на равенство значения самому себе вернёт *ложь* тогда и только тогда, кода операнд не-число NaN. Проверка на неравенство вернёт *истину*, когда хотя бы из операндов не-число NaN.

JAR и манифесты

Статей про создание JAR и его приложений много, но бывает, что трудно понять даже простое, если ни разу не видели как это делается.

Приведу очень простой пример создания Jar-архива. Примеры будут в папке `examples/manual-1/jar`

Для начала создадим нашу программу и назовем её условно Lab.java:

```java
public class Lab {
    public static void main(String ... args) {
        System.out.println("it works on my machine");
    }
}
```

Скомпилируем:

```
javac Lab.java
```

Получим байт-код `Lab.class`.

Чтобы запаковать его используем команду `jar` с параметрами `c` (create) и `f` (file).

```
jar cf Labpack.jar Lab.class
```

На выходе вы получите файл Labpack.jar, который запускается командой:

```
java -jar Labpack.jar
```

Но программа не исполнится как задумано, а вместо этого выведет:

```
no main manifest attribute, in .\Labpack.jar
```

Дело в том, что в таком jar-пакете может быть несколько файлов и исполняющая машина не может знать какую из них выполнить.

Чтобы указать ему наш класс для выполнения создадим файл `MANIFEST.MF`.

Есть несколько способов для Unix-подобных систем. Пользуйтесь таким какой вам удобнее

Вот внутренности MANIFEST.MF

```
Version: 6.6.6
Main-Class: Lab
```

Здесь мы указали версию нашего пакета, а самое главное точку входа (класс Lab)

Команда теперь будет выглядеть следующим образом:

```
jar cfm Labpack.jar MANIFEST.MF Lab.class
```

Здесь важно соблюдать порядок:

1. Имя jar-пакета
2. Путь к манифесту
3. Классы

Запустив его предыдущей командой `java -jar Labpack.jar` мы получим:

```
it works on my machine
```

Остальные контрольные вопросы

Мне не очень-то уж и хочется рассказывать про синтаксис языка Java - да, в некоторых из них есть не очевидные на первый взгляд нюансы, но проблем с базовыми вещами (for, for-each, while, do-while) быть не должно (тысячи и тысячи статей).

Поэтому я оставлю тут вопросы, на которые лучше знать ответ, и которые могут служить ориентиром:

```
1. Что такое JAR, для чего он нужен?
1.1 Что такое манифест, основные параметры (Main-Class, Version ...)
1.2 Как собирать JAR (консольная команда)

2. Чем отличаются (что такое) JDK, JRE и JVM
2.1 Что такое IDE?

3. Что такое компиляция? Какие виды еще есть (интерпретация, трансляция... (на будущее))?
3.1 Жизненный цикл Java программы

4. Почему программу написанную на Java можно запускать на любой платформе?
4.1 Какую роль в этом играет JVM?
4.2 Что такое байт-код?

5. Примитивные типы данных. Сколько их существует? Представление данных и их классификация
5.1 В чем различия double и float?
5.2 Сколько бит ответедено под каждый примитивный тип?
5.3 Константы (final). Можно ли не инициализировать константу при объявлении?
5.4 Преобразование и приведение типов. В чем отличие?
5.4 (Сложнаааа) В чем отличие примитивных и ссылочных типов данных?
5.5 В чем ошибка: byte a = 1, b =2; byte c = a + b;

6. Основные методы библиотеки Math
6.1 Вспомогательные пакеты Java. Пакет java.util (для чего он нужен).
6.1 Как хранятся эти пакеты?

7. Что такое NaN? В каких случаях он возникает и с какими типами данных?

8. Зарезервированные лексемы (continue, for, break, else ...)

9. Комментарии 3 вида (чучуть про джавадок)

10. Линейные и нелинейные программы.
10.1 синтаксис while, do-while, switch, тернарный оператор, foreach-цикл
10.2 Что вы можете рассказать о конструкции: for(;;) { }
10.3 Какие типы данных могут быть использованы в операторе switch?
10.4 Как работает тернарный оператор?

11. Стандартные потоки (out, err, in)
11.1 Как использовать стандартные потоки ввода/вывода в Java?

12. java.util.Arrays - работа с массивами
12.1 Как устроены многомерные массивы?
12.2 Сколько элементов может хранить массив?

```

```
13. (Сложнааа) оператор "new"

14. Работа со строками (trim, replace ...). Тип данных String

15. (Сложнаааа, ни нада) goto в Java. Как ставить label?
```

Приложение А. AOT- и JIT-компиляция

В этом разделе мы обсудим и сравним раннюю (Ahead-of-Time - AOT) компиляцию и компиляцию оперативную (Just-in-Time - JIT)

AOT

Если у вас есть опыт программирования на таких языках, как C или C++, то вы знакомы с AOT-компиляцией (возможно, вы всегда называли ее просто "компиляцией"). Это процесс, при котором внешняя программа (компилятор) принимает исходный текст (в удобочитаемом для человека виде) и на выходе дает непосредственно **исполняемый машинный код**.

> Ранняя компиляция исходного кода означает, что у вас есть только одна возможность воспользоваться преимуществами любых потенциальных оптимизации

Скорее всего, вы захотите создать исполняемый файл, предназначенный для конкретной платформы и архитектуры процессора, на которой вы собираетесь его запускать. Такие тщательно настроенные бинарные файлы смогут использовать любые преимущества процессора, которые могут ускорить работу программы.

Однако в большинстве случаев исполняемый файл создается без знания конкретной платформы, на которой он будет выполняться. Это означает, что AOT-компиляция должна делать консервативное предположение о том, какие возможности процессора могут быть доступны . Если код скомпилирован в предположении доступности некоторых возможностей, а затем все оказывается не так, этот бинарный файл не будет запускаться совсем .

Это приводит к ситуации, когда AOT-скомпилированные бинарные файлы не в состоянии в полной мере использовать имеющиеся возможности процессора.

JIT

Оперативная компиляция ("в точный момент времени") - это общая технология, когда программы преобразуются (обычно из некоторого удобного промежуточного формата) в высоко оптимизированный машинный код непосредственно во время выполнения.

HotSpot и большинство других основных производителей JVM в значительной степени полагаются на применение этого подхода. При таком подходе во время выполнения собирается информация о вашей программе и создается профиль, который можно использовать для определения того, какие части вашей программы используются наиболее часто и больше всего выиграют от оптимизации.

Подсистема JIT использует ресурсы VM совместно с вашей запущенной программой, поэтому стоимости такого профилирования и любых выполняемых оптимизаций должны быть сбалансированы с ожидаемым прирос ом производительности.

Стоимость компиляции байт-кода в машинный код платится во время выполнения; компиляция расходует ресурсы (процессорное время, память), которые в противном случае могли бы быть использоваться для выполнения вашей программы.

Поэтому JIT-компиляция выполняется экономно, а VM собирает статистику о вашей программе (ищет "горячие пятна"), чтобы знать, где лучше всего выполнять оптимизацию.

Приложение Б. Использованная литература

- Tim Lindholm, Frank Yellin - JVM Specification Java SE8 Edition
- Брюс Эккель - Философия Java
- Benjamin Evans - Optimizing Java. Practical techniques for improving JVM application performance
- Письмак А.Е. - Конспекты лекций первого семестра
- Э. Таненбаум - Архитектура компьютера
- Ну и как всегда `stackoverflow`

ООП в контексте Java (редактируется)

Весь материал, который Вы понимаете, сразу применяйте на практике. Придумывайте идеи и старайтесь реализовывать, используя то, чему научились на занятиях или при самостоятельном изучении.

Письмак

Объектно-Ориентированное Программирование

Мини-вступление

Чтобы понять "Что такое ООП?" мне понадобилось 3 недели (если не считать, что задолго до этого пытался изучить ООП в контексте C# и Python), я уже мог пользоваться этими объектами и использовать в своем Java-коде (к слову, это была усложненная версия первой лабы).

Но, по крайней мере, я так думал. На то, чтобы **действительно** понять всю суть ООП, у меня ушло намно-оо-го больше времени, и я до сих пор думаю, что не до конца понимаю ООП.

К чему я это? К тому, что читая какую-нибудь статью - да, вы будете знать как использовать ООП в разных языках, но не сможете ощутить всю их прелесть не покодив порядочное количество проектов. В этом я согласен с Письмаком и полностью поддерживаю его слова.

Для "опытных" пользователей компьютера

Если вы уже знаете что такое ООП на уровне свободного использования объектов на любом языке поддерживающим парадигму ООП, то сразу можете переходить к главе **"ООП в Java"**

Далее, мы рассмотрим ООП в общем плане, не привязывая чисто к Java, но для примеров будем использовать его.

Канон

Обычно, люди когда объясняют про всякие объекты, классы, их методы и тд, они начинают с класса. Ну, это вполне логично, потому что **объект создается из класса**.

Но я поступлю иначе, и сначала попытаюсь объяснить "что такое объект?".

Давайте попробуем связать его определение с тем, что мы уже знаем.

Кузнечик

Представим себе кузнечика. Пусть, это будет наш объект.

Какие у него есть свойства? Например, длина, окрас и пускай у него еще будет имя Боб.

Итак, попробуем записать нашего Боба:

```
Grig(кузнечик){
    length: 5;
    color: brown;
    name: "Bob";
}
```

Прекрасно, а что он умеет делать? Скажем, например, прыгать. Давайте запишем это как функцию:

```
Grig{
    length: 5;
    color: brown;
    name: "Bob";

    function jump(){ "jump 25 cm" };
    function eat(){ "eat green grass" };
}
```

Теперь, у нас есть объект - у него свойства (length, color, name) и методы (действия в данном случае)(jump, eat). Здесь важно понимать, что jump и eat - это функции, то есть выполняют какую-то операцию.

Но, насколько бы он не был интровертом, думаю, ему все равно нужна пара, поэтому давайте создадим ему девушку:

```
Grig{
    length: 4.9;
    color: green;
    name: "Alice";

    function jump(){ "jump 25 cm" };
    function eat(){ "eat green grass" };
}
```

Когда у нас есть два объекта, попробуем сравнить их. У них те же имена свойств (length, color, name), но разные значения. В том числе, у них одинаковые имена методов (jump, eat).

А если у них имена всех свойств и методов совпадают - давайте сделаем какой-то шаблон, чтобы из него создавать эти объекты. Пусть, это будет шаблон с именем Grig и будем создавать эти объекты по этому шаблону. При этом функции везде одинаковые, поэтому пусть это сразу будет в шаблоне. Тогда нам нужно будет указать лишь уникальные свойства.

```
class Grig {
    length: null;
    color: null;
    name: null;

    function jump(){ "jump 25 cm" };
    function eat(){ "eat green grass" };
}
```

Как видите, мы не можем знать какие свойства будут у объекта, поэтому просто поставим там значения `null` . Отсюда и можно понять, что **класс - это не объект**, а сущность от которого эти объекты создаются.

Мы представили объект и класс как кузнечиков. А теперь попробуйте посмотреть вокруг себя внутри комнаты, на улице. Все является объектом! И ведь правда, любой встреченный человек - это объект из шаблона (класса) человек. Или, например, лампа - у нее есть свой цвет, размер (свойства), к тому же она может светить (метод).

Здесь я бы хотел привести цитату, которую повторял мой учитель информатики. Она, вроде как я помню, была от Брюса Ли, а оригинал я не нашел, но суть была такая:

> Видеть Кунг-Фу во всем

Казалось бы не совсем понятная цитата и я сначала посмеялся, но мой учитель объяснил, что "кунг-фу" это образное выражение того, чем ты занимаешься. Ну, на тот момент эта цитата изменилась на "видеть программирование во всем, что нас окружает".

И вот однажды после пар на Кронверкской я направлялся на Горьковскую и неожиданно меня осенило:

-- "Так вот же объекты, вот они проходят мимо меня, эти чертовы люди! Вот стоит машина, а ведь это тоже объект".

Так, восприятие моего мира изменилась, хотя на жизнь это вряд ли повлияло.

ООП в Java

В этой главе рассмотрим парадигмы ООП в контексте Java, потому что без конкретных примеров объяснить будет очень трудно, но следует заметить, что **парадигмы ООП встречаются во многих языках**, но имеют свою реализацию.

Предисловие: если вы понимаете парадигмы ООП и умеете применять их в стеке Java, то сразу можете переходить к главе со звездочкой **Стек и Куча**.

Итак, погружаемся в ООП...

Прежде чем приступить к парадигмам, научимся создавать объекты и классы в Java.

Примеры исходников можно найти в `examples/manual-2`

Давайте реализуем наши классы кузнечиков в контексте Java:

Листинг 1.1 FirstExample.java

```java
class Grig {
    double length;
    String color;
    String name;

    void jump(){
        System.out.println(name + " is jumping");
    }

    void eat(){
        System.out.println(name + " is eating grass");
    }
}

public class FirstExample {
    public static void main(String[] args) {

        // создаем объект-кузнечик Bob
        Grig bob = new Grig();
        bob.name = "Bob";
        bob.length = 5d;
        bob.color = "brown";

        // создаем объект-кузнечик Alice
        Grig alice = new Grig();
        alice.name = "Alice";
        alice.length = 4.9d;
        alice.color = "green";


        bob.jump();     // output: Bob is jumping
        alice.jump();   // output: Alice is jumping

        bob.eat();      // output: Bob is eating grass
        alice.eat();    // output: Alice is eating grass

    }

}
```

Очень надеюсь, что мне не стоит объяснять как обращаться к свойствам (полям) или методам объекта. Так как вы, скорее всего, уже их использовали, например так:

```java
System.out.print("hello, onii-chan!")
```

Не сказал бы, что это совсем удачный пример, так как здесь затрагиваются статические поля, но не суть. Сначала вы обращаетесь к `out` и затем от него вызываете метод `print()`

По сути, `out` это **статическое** поле в классе `System` , который имеет несколько методов, в том числе и `print` (что такое статическое разберем позже)

Вернемся к листингу и разберем все по полочкам:

1. Сначала мы **объявляем переменную** `bob` с типом `Grig`. Замечаете определенные сходства со `String` (тут же и вопрос почему `String` нужно сравнивать через `equals`)?
2. А затем нам нужно **создать объект** из класса кузнечика и **присвоить** это **значение** к нашей переменной `bob`. Если с присвоением все понятно, то как создавать объекты из класса? По сути, также как и массивы - через оператор `new` . Его мы тоже разберем чуть позже.
3. Теперь как вы помните класс не может быть объектом, поэтому его поля не инициализированы то есть не имеют значений или же если быть корректнее - имеют значения по умолчанию. Если вы не помните или не знаете значения по умолчанию посмотрите предыдущий мануал.
   Значит, их нужно инициализировать, а сделали мы это очень тривиально и понятно.
4. Теперь можем попробовать вызвать методы класса, общаясь к ним через объекты (экземпляры)

Конструкторы

Согласитесь, неприятно и в общем-то неудобно задавать поля (свойства) класса вот так:

```
Grig bob = new Grig();
bob.name = "Bob";
bob.length = 5d;
bob.color = "brown";
```

Тут на помощь к нам приходят конструкторы. Давайте сначала посмотрим его реализацию, а затем разберемся что к чему:

Листинг 1.2 SecondExample :

```
public class SecondExample {
    static class Grig {
        double length;
        String color;
        String name;

        Grig(String grigsName, String grigsColor, double grigsLength){
            name = grigsName;
            color = grigsColor;
            length = grigsLength;
        }

        void jump(){ System.out.println(name + " is jumping"); }
        void eat(){ System.out.println(name + " is eating grass"); }
    }

    public static void main(String[] args) {
        // создаем объект-кузнечик Bob
        Grig bob = new Grig("Bob", "brown", 5d);

        // создаем объект-кузнечик Alice
        Grig alice = new Grig("Alice", "green", 4.9d);
```

```
23
24
25          bob.jump();     // Bob is jumping
26          alice.jump();   // Alice is jumping
27          bob.eat();      // Bob is eating grass
28          alice.eat();    // Alice is eating grass
29      }
30  }
```

В основном все также, но сравните предыдущий пример инициализации полей, и вот такую:

```
1  Grig bob = new Grig("Bob", "brown", 5d);
2
```

Здесь было бы уместно сказать : "*Краткость - сестра таланта*"

> Не обращайте внимания на модификатор static перед объявлением класса, сейчас это к делу не относится! Вернемся к ней позже

Итак, если посмотреть изменения, то мы добавили что-то похожее на функцию, которое имеет такое же имя как у класса и к тому же не имеющий типа возвращаемого значения (даже `void` здесь не видно!):

```
1  Grig(String grigsName, String grigsColor, double grigsLength){
2      name = grigsName;
3      color = grigsColor;
4      length = grigsLength;
5  }
6
```

Но понять, что именно он делает мы можем - берем значения из аргументов нашей "псевдо-функции" и присваиваем их соответственно по значениям полей. Все просто!

Думаю, вы уже догадались откуда мы будем получать эти аргументы - при вызове `new Grig()`

Так, `Grig()` - это метод или нет? Попробуйте использовать `Grig()` как обычный метод :)

У вас будет ошибка компиляции, потому что, `Grig()` - это действительно метод (это можно сказать по его схожести объявления в классе), но как вы могли заметить - **он особенный** .

Если коротко, то:

> Конструктор - это специальный метод, который вызывается при создании нового объекта

Подождите! Мы же их вызывали раньше, а там ведь не было никаких методов!

```
1  Grig bob = new Grig();
2
```

Да, если попробовать запустить его, добавив конструктор такого вида (который ничего не делает):

```
Grig(){

}

```

Код все равно будет рабочим, а это значит, что если созданный вами класс не имеет конструктора, **компилятор автоматически добавит конструктор по умолчанию**.

Это можно увидеть в байт-коде через команду `javap` (пример из предыдущего мануала):

```
public class Hello {
        public static void main(String ... args){
                for(int i=0;i<10;i++){
                        System.out.println("ITMO");
                }
        }
}

```

Компилим и смотрим его байт-код:

```
Compiled from "Hello.java"
public class Hello {
  public Hello();       // <--- вот эта вот!
    Code:
       0: aload_0
       1: invokespecial #1    // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String...);
    Code:
       0: iconst_0
       1: istore_1
       2: iload_1
       3: bipush        10
       5: if_icmpge     22
       8: getstatic     #2    // Field
java/lang/System.out:Ljava/io/PrintStream;
      11: ldc           #3    // String ITMO
      13: invokevirtual #4    // Method java/io/PrintStream.println:
(Ljava/lang/String;)V
      16: iinc          1, 1
      19: goto          2
      22: return
}

```

Мы не объявляли никакого конструктора, но в скомпилированной версии он есть. Его также именуют "***конструктором по умолчанию***" или в документации Java "*no-arg constructor*"

Следует заметить, что создание хотя бы одного конструктора уже отменяет автоматическое добавление конструктора по умолчанию.

Идем дальше, так в чем же заключается **особенность конструктора**?

Во-первых, **имя конструктора должно совпадать с именем класса**. Причиной этому стали две тонкости:

1. Любое имя, которое вы используете, может быть задействовано при определении членов класса, так возникает *конфликт имен*.
2. *За вызов конструктора отвечает компилятор*, поэтому он всегда должен знать, какой именно метод следует вызвать.

Во-вторых, **у конструктора отсутствует возвращаемое значение**. Конструкторы никогда и ничего не возвращают (оператор `new` возвращает ссылку на вновь созданный объект, но сами конструкторы не имеют выходного значения).

К слову, **в классе может быть несколько конструкторов**, но они как и методы, должны иметь разную сигнатуру (входные аргументы).

Также обязательно посмотрите главу **Блоки инициализации**

Стек и куча

В этой главе речь пойдет о хранении данных в Java, в том числе про стек(stack) и кучу(heap). И перед тем как приступить к этой главе, я бы настоятельно рекомендовал получше изучить объекты и их работу с ними. Попробуйте, например, воссоздать объекты из реального мира.

Самое главное, вам нужно понять как работать с ними.

Далее, идет глава не самая легкая для понимания. Если вы впервые сталкиваетесь с ООП, то тем более. Но это не говорит о том, что эту главу можно пропустить. Почитайте. Таким образом, вы восполните свой словарный запас и хотя бы на каком-то (очень абстрагированном) уровне поймете принцип работы ООП в Java.

Когда-то, я начал читать книгу Джоша Лонга "*Java EE для предприятий*" . Там рассказывалось про архитектуру приложений в Java EE, но не суть. Дело в том, что мой уровень не позволял понять полностью, о чем в этой книге говорится, но я все равно читал.

И когда я одновременно с этим листал презентацию из se ifmo или из других источников, то сразу вспоминал слова находящиеся там и мог примерно понять, о чем идет речь.

К слову, эту книгу я перечитывал трижды и каждый раз одни и те же главы открывали для меня что-то новое, что я не мог увидеть раньше.

Ссылочные типы

Виртуальная машина Java содержит явную поддержку объектов. Объектом мы называем динамически создаваемый экземпляр класса или массив. Ссылка на объект представлена в виртуальной машине Java типом `reference`. Значения типа `reference` могут быть рассмотрены как указатели на объекты. На один и тот же объект может

существовать множество ссылок. Передача объектов, операции над объектами, проверка объектов происходит всегда посредством типа `reference`.

Так, `bob` и `alice` (наши переменные) являются ссылками на объекты.

Существуют три разновидности ссылочных (`reference`) типов: **тип класса, тип массива и тип интерфейса**. Значения этих типов представляют собой ссылки на экземпляр класса, ссылки на массив и ссылки на имплементацию интерфейса соответственно (про интерфейсы еще далеко).

Тип массива представляет собой *составной тип* единичной размерности (длина которого не определена типом). Каждый элемент составного типа сам по себе может также быть массивом. Последовательно рассматривая иерархию составных типов в глубину, (тип, из которого состоит составной тип, из которого состоит составной тип и т.д.) мы придём к типу, который не является массивом; он называется *элементарным типом* типа массив. Элементарный тип всегда либо примитивный тип, либо тип класса, либо тип интерфейса.

Тип `reference` может принимать специальное нулевое значение, так называемая ссылка на не существующий объект, которое обозначается как `null`. Значение `null` изначально не принадлежит ни к одному из ссылочных типов и может быть преобразовано к любому.

Конструкторы копирования

Конструкторы копирования - это не специально предусмотренные конструкторы, а лишь общее название их функциональности, а именно копирования полученного объекта.

Рассмотрим пример:

```
class Point {
    private String name;
    public void setName(String name){
      this.name = name;
    }

    public String getName(){
      return this.name;
    }
}

// ...

Point p1 = new Point();
p1.setName("v1");

Point p2 = p1;
p2.setName("v2");

System.out.println(p1.getName()); // output: v2
System.out.println(p2.getName()); // output: v2
```

Изменение значения переменной `name` в `p2` вызвало изменения в `p1`. Почему же это произошло?

Все из-за того, что `p1` и `p2` это ссылки на действительный объект, поэтому когда мы выполняем операцию присваивания со ссылками, то просто назначаем ссылку на объект. Если смотреть более детально, то сначала мы создали ссылку на объект `Point` с именем `p1`. Затем ссылке `p2` присвоили ссылку `p1`. Теперь они оба указывают на один и тот же объект, в следствие чего изменения состояния через одну ссылку, несут изменения на другую.

Чтобы таких случаев не было, обычно создают так называемые "конструкторы копирования" - это такие конструкторы, которые на вход получают объект своего же класса и создают идентичный объект. Здесь существенное отличие в том, что при присваивании мы должны будем вызвать `new` в следствие чего создастся новый объект, а не ссылка.

Чтобы реализовать такой конструктор, нам достаточно присвоить состояние входного объекта к нашему:

```
class Point {
    public Point() {}

    public Point(Point p){
      // конструктор копирования
      this.name = p.getName();
    }

    private String name;
    public void setName(String name){
      this.name = name;
    }

    public String getName(){
      return this.name;
    }
}
```

В конструкторе копирования, мы присваиваем все поля входного объекта в соответствующие поля создаваемого объекта. Таким образом, мы можем изменить предыдущий код:

```
Point p1 = new Point();
p1.setName("v1");

Point p2 = new Point(p1);

p2.setName("v2");

System.out.println(p1.getName()); // v1
System.out.println(p2.getName()); // v2
```

Здесь нужно быть осторожным, особенно когда у вас много полей.

Например, вы можете забыть инициализировать некоторые поля, из-за чего вы получите не копию объекта, а другой.

Для решения такой проблемы есть несколько путей.

Во-первых, вы можете ссылаться на `this()` , то есть другой конструктор встроенный в ваш класс. Здесь вас может постигнуть неудача, если нету конструктора, который инициализирует все поля. Тогда вам все равно придется присваивать значения некоторым оставшимся полям вручную.

Во-вторых, если все ваши поля имеют модификатор `final` (разберем в следующих главах). Вам просто придется инициализировать эти поля в конструкторе, таким образом, если вы допустите ошибку - компилятор вам об этом скажет.

Области данных в рантайме

Регистр PC

Виртуальная машина Java может поддерживать множество потоков, выполняющихся одновременно. Каждый поток виртуальной машины Java имеет свой регистр **pc** (*program counter*). В каждый момент времени каждый поток виртуальной машины исполняет код только одного метода, который называется текущим методом для данного потока. Если метод платформенно независимый (т.е. в объявлении метода не использовано ключевое слово `native`) регистр pc содержит адрес выполняющейся в данный момент инструкции виртуальной машины Java. Если метод платформенно зависимый (`native` метод) значение регистра **pc** не определено.

Стек виртуальной машины Java

Каждый поток виртуальной машины имеет свой собственный *стек виртуальной машины Java*, создаваемый одновременно с потоком. Стек виртуальной машины хранит **фреймы**.

Стек виртуальной машины Java аналогичен стеку в традиционных языках программирования: он хранит локальные переменные и промежуточные результаты и играет свою роль при вызове методов и при возврате управления из методов. Поскольку работать напрямую со стеком виртуальной машины Java запрещено (кроме операций `push` и `pop` для фреймов), фреймы могут быть также расположены в куче. Участок памяти для стека виртуальной машины Java не обязательно должен быть непрерывным.

В следующих случаях виртуальная машина Java формирует исключение при работе со стеком:

- Если вычисления в потоке требуют памяти более чем позволено размером стека, виртуальная машина Java формирует исключение `StackOverflowError`.
- Если стек виртуальной машины Java допускает динамическое увеличение размера и попытка такого увеличения была выполнена, однако вследствие нехватки памяти не завершена успешно либо не достаточно памяти при инициализации стека при создании потока, то виртуальная машина Java формирует исключение `OutOfMemoryError`.

Не будем подробно разбирать исключения, а это совсем отдельная тема, но если вы встретите их, то уже будете знать в чем дело (хотя и не факт, что сможете пофиксить)

Куча

Виртуальная машина Java содержит область памяти, называемую *кучей*, которая находится в пользовании всех потоков виртуальной машины. Куча – это область памяти времени выполнения, содержащая массивы и экземпляры всех классов.

Куча создаётся при запуске виртуальной машины Java. Удаление неиспользуемых объектов в куче производится системой автоматического управления памятью (известной как *сборщик мусора* (к этой теме мы еще вернемся))

Объекты никогда не удаляются явно. Виртуальная машина Java не предполагает какого-либо одного алгоритма для системы автоматического управления памятью; алгоритм может быть произвольно задан разработчиком виртуальной машины в зависимости от системных требований. Куча может быть фиксированного размера, либо динамически расширяться и сужаться при удалении объектов.

Участок памяти для кучи виртуальной машины Java не обязательно должен быть непрерывным.

Что и где хранится?

Мы рассмотрели два хранилища данных программы - стек и куча. Почему же их две и чем они отличаются?

Сначала, рассмотрим что такое стек и как она работает.

Во-первых, под стеком подразумевается некоторый принцип хранения данных и обращения к данным. Обычно здесь можно привести в пример стопку тарелок. Мы можем положить тарелку сверху и взять тоже только сверху. Такой принцип называется LIFO (Last-In-First-Out).

Согласитесь, довольно странный способ хранения информации, правда?

Давайте разберемся как он может пригодиться в нашей программе.

> Далее, будет много байт-кода и углубление в JVM, но как по мне - это не так сложно понять, хотя и будут трудности. И эту главу необязательно читать, если вы желаете сдать только лабу, но если вы хотите больше узнать как работает ваша любимая JVM - добро пожаловать!
>
> И также, обязательно к прочтению глава `this` - это просто необходимо знать.

Зачем нужен стек в Java?

Давайте внесем немножко Java в свою жизнь:

Листинг 2.1 DataExample.java

```
1  public class App {
2   public static void main(String[] args) {
3       System.out.print("Enter a:");
4       int a = Integer.parseInt(System.console().readLine());
5       System.out.print("Enter b:");
6       int b = Integer.parseInt(System.console().readLine());
7       System.out.println("a+b=" + pow(a,b));
8   }
9   public static Long pow(int base, int exponent) {
10      Long result = 1L;
11      for (int i = 0; i < exponent; i++) {
12           result *= base;
13      }
```

```
    return result;
 }
}

```

Давайте сначала рассмотрим только метод main.

Какие у нас данные? Во-первых, это наши переменные `a`, `b`. Также, у нас входные данные `args`. Итого, мы насчитали 3, давайте это проверим, запустив команду:

```
javap -v -c DataExample

```

Пропускаем оттуда пул констант и переходим сразу к этому:

```
  public DataExample();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."
<init>":()V
         4: return
      LineNumberTable:
        line 4: 0

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=4, locals=3, args_size=1    // <----------- [ ЗДЕСЬ ] !!!
         0: getstatic     #2
         3: ldc           #3         // String Enter a:
         5: invokevirtual #4
         8: invokestatic  #5
        11: invokevirtual #6
        14: invokestatic  #7
        17: istore_1
        18: getstatic     #2
        21: ldc           #8         // String Enter b:
        23: invokevirtual #4
        26: invokestatic  #5
        29: invokevirtual #6
        32: invokestatic  #7
        35: istore_2
        36: getstatic     #2
        39: new           #9          // class java/lang/StringBuilder
        42: dup
        43: invokespecial #10
        46: ldc           #11         // String a+b=
        48: invokevirtual #12
        51: iload_1
        52: iload_2
        53: invokestatic  #13         // Method pow:(II)Ljava/lang/Long;
        56: invokevirtual #14

```

```
        59: invokevirtual #15
        62: invokevirtual #16
        65: return

     public static java.lang.Long pow(int, int);
     // к методу pow мы вернемся позже
}
SourceFile: "DataExample.java"

```

Давайте обратим внимание на строку : `stack=4, locals=3, args_size=1`. Как видно из неё, мы оказались правы - 3 локальные переменные, в том числе одна из них это входные данные

Разберем значения этих трех свойств:

- `stack` - максимальный размер стека, необходимый для выполнения процедуры
- `locals` - локальные переменные
- `args_size` - входные данные. Есть парочка интересных моментов с ней, поэтому мы еще вернемся к этому параметру.

Теперь посмотрим на метод `pow` :

```
  public static java.lang.Long pow(int, int);
    descriptor: (II)Ljava/lang/Long;
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=4, locals=4, args_size=2
         0: lconst_1
         1: invokestatic  #14
         4: astore_2
         5: iconst_0
         6: istore_3
         7: iload_3
         8: iload_1
         9: if_icmpge     29
        12: aload_2
        13: invokevirtual #15
        16: iload_0
        17: i2l
        18: lmul
        19: invokestatic  #14
        22: astore_2
        23: iinc          3, 1
        26: goto          7
        29: aload_2
        30: areturn
      LineNumberTable:
        line 10: 0
        line 11: 5
        line 12: 12
        line 11: 23
        line 14: 29
      StackMapTable: number_of_entries = 2
        frame_type = 253 /* append */
```

```
          offset_delta = 7
          locals = [ class java/lang/Long, int ]
        frame_type = 250 /* chop */
          offset_delta = 21

```

Здесь для выполнения операций нужно четыре слота стека

Также, мы имеем два аргумента и еще две локальные переменные - `result` и `i`

Задачка для тех, кто умеет работать с ООП

```
public class Test {
   public void plus(){
        int a = 1;
        int b = 3;
   }
 }
// stack=1, locals=3, args_size=1


```

Разберите в этом примере, почему в стеке операндов нам нужен лишь один слот и почему у нас три локальных переменных. (*Подсказка*: args_size тоже входит в их число)

Смотрите сюда, если вы не смогли узнать сами почему там `args_size = 1` или сделали какое-нибудь предположение, следующая подсказка:

```
public class Test {
   public static void plus(){
    int a = 1;
    int b = 3;
   }
 }
// stack=1, locals=2, args_size=0


```

Ответ

Во-первых, у нас есть две переменные `a`, `b` - уже две локальные переменные, а где же третья?

Третья переменная - это `this` (ссылка на экземпляр класса, если вы не знаете, что это, то разберем дальше). Вам могла помочь последняя подсказка, когда мы объявили метод `plus` статической.

Теперь, почему же в стеке нам нужен лишь один слот? - потому что нам нужно только сохранить 1 и присвоить переменной, затем нам эта единица в стеке уже не нужна, поэтому мы можем заменить её уже тройкой, а еще один слот нам уже не нужен.

Таинственный this и static

Что же это за `this`, который передается нестатическим методам?

`this` - это ссылка на экземпляр класса. Иначе, если мы создаем объект из нашего класса, то мы можем ссылаться на его экземпляр через переменную `this`, поэтому в байт-коде, как вы могли видеть, присутствует `args_size=1`.

Зачем нам это нужно?

Во-первых, мы можем ссылаться на поля своего же объекта. Да, можно использовать только имя поля - это разрешено, но иногда бывают ситуации, когда имя аргумента функции совпадает с полем класса. Возникает конфликт имен - ошибки компиляции, конечно, не будет, но вы не сможете обратиться к полю класса.

Например, первый наш пример можно заменить таким образом:

```
public class SecondExample {
    static class Grig {
        double length;
        String color;
        String name;

        Grig(String name, String color, double length){
            this.name   = name;
            this.color  = color;
            this.length = length;
        }

        // some methods
    }

    public static void main(String[] args) {
        // something actions
    }
}

```

Такой код читается лучше и понятнее.

Разумеется, это не единственное предназначение ссылки `this`

Второе и очень важное его предназначение, а точнее способ его использования - это возможность передавать ссылку на экземпляр. Звучит непонятно...

Допустим, у вас есть метод который принимает пользовательский тип:

```
public void func(MyClass clazz){ /* something actions */};

```

Использовать её не составляет труда:

```
MyClass ex = new MyClass();
func(ex);

```

Но вдруг мы хотим использовать её внутри реализации нашего класса.

То есть, у нас есть некоторый метод в MyClass, который должен вызывать функцию `func` , передавая экземпляр самого себя. Без `this` это практически было бы нереализуемо, по крайней мере, очень сложно.

А здесь мы можем использовать:

```
public MyClass{
    public void myMethod(){
        func(this);
    }
}

```

Возвращаясь к хранениям данных, то мы уже знаем, что этому методу неявно передается ссылка на `this` (см. главу "Зачем нужен стек")

Теперь о статических методах - это методы, которые прикреплены не к экземпляру класса, а к самому классу. То есть они не требуют экземпляра, чтобы выполниться и при этом, им не передается ссылка на `this` . Опять же мы могли видеть это в предыдущей главе.

Их преимущество в том, что они не требуют экземпляра класса, поэтому мы можем использовать такой код:

```
class MyClass {
    public static void func(int a){ /*something action...*/};
}

// еще где-то в исходниках:
MyClass.func(12);

```

Как видите, мы не создавали экземпляр класса MyClass, а сразу обратились к методу.

Статические поля или методы объявляется с ключевым словом `static`

Обоюдоострым мечом статических методов является то, что статические данные одни и те же для всех экземпляров. То есть, если у нас есть два экземпляра класса и каждый из них увеличит статическое поле класса на единицу, то в итоге мы получим +2, так как данные едины для всех экземпляров класса. Это очень логично так как статические методы и поля принадлежат классу, а не экземпляру класса.

Давайте рассмотрим пример:

Листинг 3.1 StaticExample.java

```
public class StaticExample {
    static class MyClassNonStatic {
        int a = 0;
        void inc(){
```

```
            a++;
        }
    }

    static class MyClassWithStatic {
        static int a = 0;
        void inc(){
            a++;
        }
    }

    public static void main(String ... args) {
        MyClassNonStatic a = new MyClassNonStatic();
        MyClassNonStatic b = new MyClassNonStatic();
        a.inc();
        b.inc();

        System.out.println(a.a + " " + b.a);    // 1 1

        a.inc();

        System.out.println(a.a + " " + b.a);    // 2 1

        MyClassWithStatic c = new MyClassWithStatic();
        MyClassWithStatic d = new MyClassWithStatic();

        c.inc();
        d.inc();
        System.out.println(c.a + " " + d.a);    // 2 2

        c.inc();
        System.out.println(c.a + " " + d.a);    // 3 3
    }
}


```

Здесь мы создаем два класса: `MyClassNonStatic` - без статического поля, `MyClassWithStatic` - со статическим полем.

Из примера все видно - наглядно и просто.

Минусом статических методов, является то, что они не могут использовать нестатические поля класса, так как они не имеют ссылки на `this` (экземпляр класса). Для более легкого понимания, просто можно думать, что для вызова нестатических методов нужен экземпляр класса, а нестатические должны использоваться и без экземпляра (упрощенно).

> Практика: попробуйте в классе MyClassNonStatic объявить метод func статическим и обратится к полю "a"

Преимущество стека и области видимости

У многих мог возникнуть вопрос, а почему мы используем стек?

Для этого рассмотрим еще один термин **области видимости.**

У каждой переменной есть область видимости - область, внутри которой можно обращаться к переменной.

Простой пример,

```
public class MyClass {
    static void func(){
        int a = 2;
    }
    public static void main(String ... args){
        func();
        System.out.println(a);  // error: cannot find symbol
    }
}

```

Здесь мы пытаемся обратится к переменной `a` из другого метода, но получаем ошибку, так как область видимости переменной `a` ограничена блоком кода функции `func`, иными словами, переменная объявленная внутри функции `func` может использоваться только внутри неё самой.

Мы увидели пример того как область переменной ограничивается блоком кода (телом функции).

Теперь посмотрим пример того, как область видимости может содержать другие области видимости:

```
public class MyClass {

    static int a = 5;

    public static void main(String ... args){
        System.out.println(a);  // 5
    }
}

```

Область видимости main содержится внутри области видимости тела класса, поэтому мы можем использовать переменную `a`.

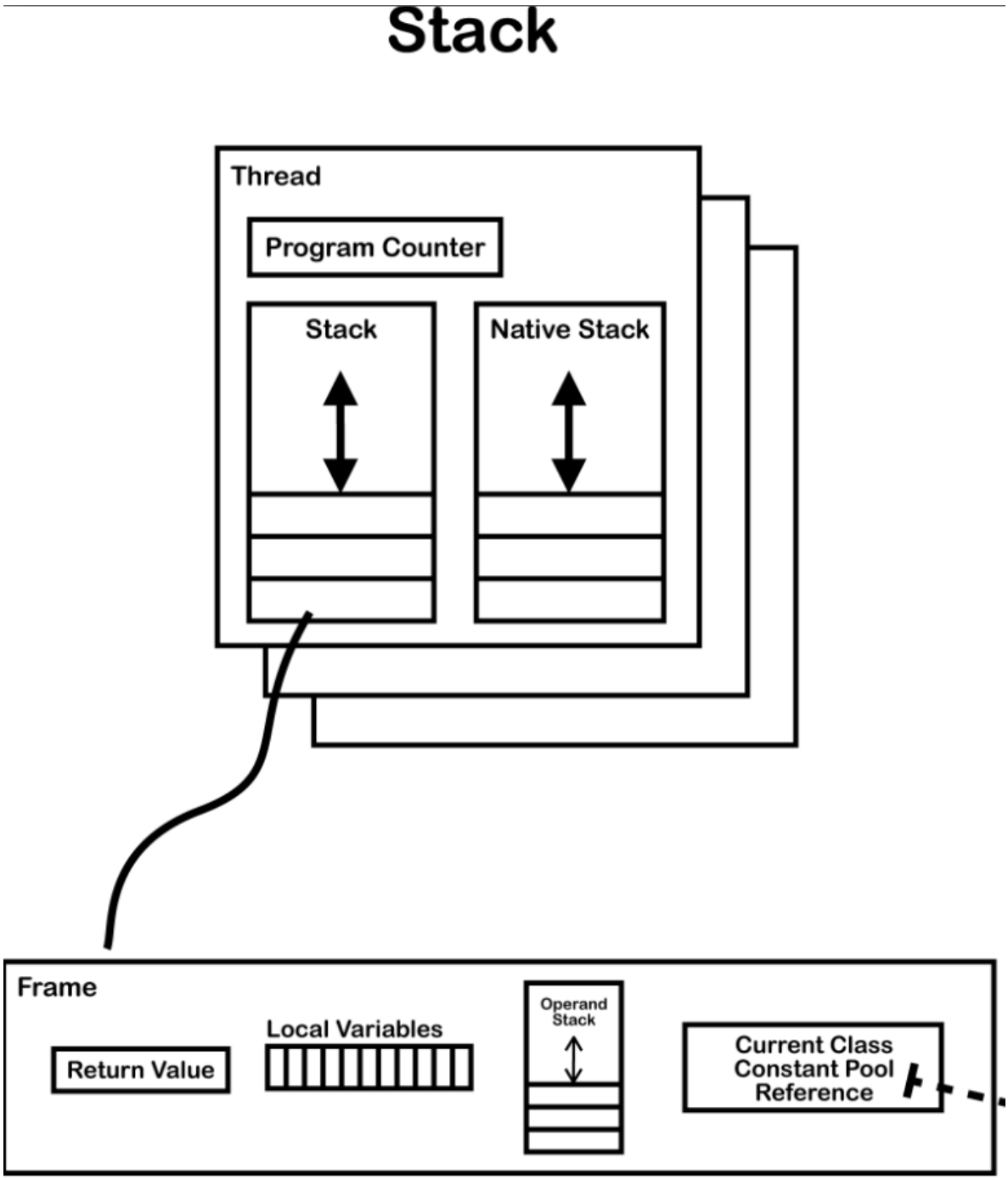
Выходит что мы не можем объявить переменную с именем `a`? Нет, мы можем её объявить, и тогда мы будет обращаться уже к ней - тут все дело в том, как JVM ищет нужную переменную.

Сначала мы ищем эту переменную в своей области видимости, где она объявлена, если мы не находим там, то ищем в той области, где находится наша область и т.д. Если мы ничего не найдем, то компилятор выведет ошибку `cannot find symbol`.

Как это все организовано?

Принцип работы со стеком очень удобно применять для управления временем жизни и видимости переменных.

Механизм такой: когда программа начинает исполнять какую-то функцию, то под используемые в ней переменные выделяется место в стеке (это не то, что указано в байт-коде в свойстве `stack`, напомню, что тот стек - это стек операндов). Для наглядности посмотрим схему:



Да-да, в потоке программы есть несколько стеков, но углубляться мы не будем, так как лишь хотим узнать как хранятся данные нашей программы.

Итак, что такое стек операндов, который мы ранее разбирали? Он используется при выполнении инструкций байт-кода.

Например, когда мы хотим объявить переменную

```
int i;

```

То получаем такую инструкцию, которая взаимодействует со стеком операндов и локальными переменными:

```
    // Положить 0 наверху стека операнодов:
    0:  iconst_0

    // вытолкнуть верхнее значение из стека операндов
    // и сохранить как локальную переменную 1
    1:  istore_1

```

После этой операции в стеке операндов не остается значений - все лишнее и не используемое в конкретный момент - убирается из стека операндов.

Поэтому в этом коде понадобится лишь один слот стека операндов:

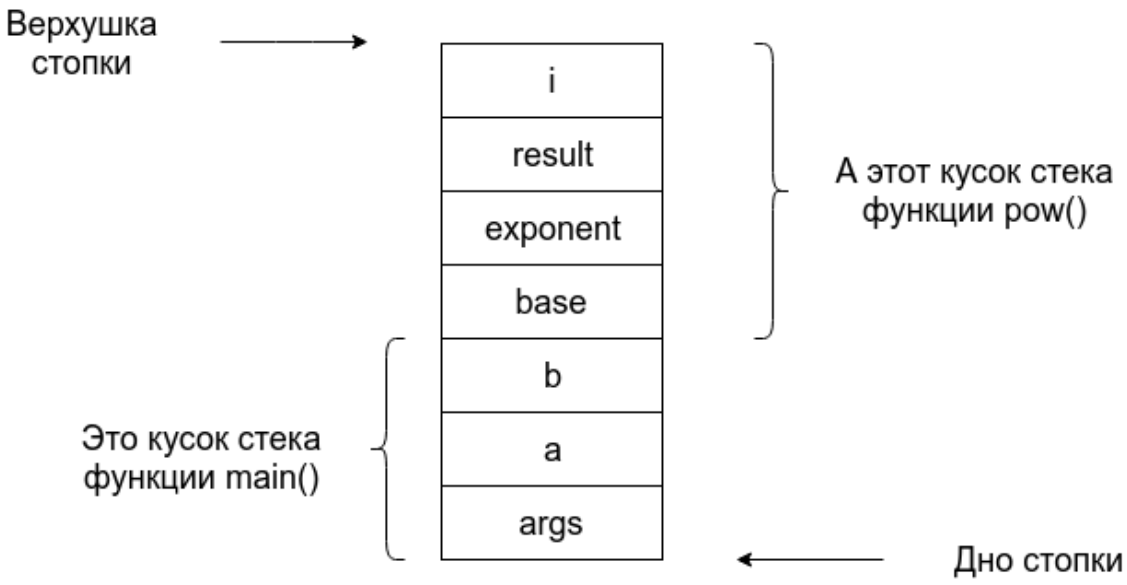
```
public class Test {
   public static void plus(){
    int a = 1;
    int b = 3;
   }
 }
// stack=1, locals=2, args_size=0

```

Возвращаемся к листингу 2.1:

Листинг 2.1 DataExample.java

```
public class DataExample {
 public static void main(String[] args) {
     System.out.print("Enter a:");
     int a = Integer.parseInt(System.console().readLine());
     System.out.print("Enter b:");
     int b = Integer.parseInt(System.console().readLine());
     System.out.println("a+b=" + pow(a,b));
 }
 public static Long pow(int base, int exponent) {
     Long result = 1L;
     for (int i = 0; i < exponent; i++) {
          result *= base;
     }
     return result;
 }
}

```

Упростим себе представление стека (не стек операндов) и посмотрим что там происходит:



При старте нашей программы в стеке будет выделено место под переменные `args`, `a`, `b`. По мере выполнения этой функции, ячейки в стеке будут заполняться какими-то значениями. В тот момент, когда программа дойдет до вызова функции `pow` в стеке создастся место под переменные необходимые для этой функции.

На рисунке мы можем разделить стек на несколько участков - это и есть *фреймы* (или иногда их называют кадрами). Заметим, что **это упрощенное представление стека**, на самом деле, там гораздо больше значений.

Здесь важно еще то, что аргументы функции (`base`, `exponent`) являются отдельными местами в памяти. Может показаться, что раз в нашей программе мы передаем в функцию `pow` `a` и `b`, то внутри неё мы будем общаться с этими местами в стеке.

Это мнение ошибочно. **Код вызываемой функции `pow` не может менять переменные внешней функции `main`**, поэтому когда при исполнении программа доходит до строки с вызовом функции `pow(a, b)`, значения, которые хранятся в соответствующих местах **копируются в места выделенные под `base` и `exponent`.** В момент, когда внутренняя функция заканчивает свою работу, то место в стеке используемое под неё очищается и может быть использовано дальше. Например, для вызова следующей функции.

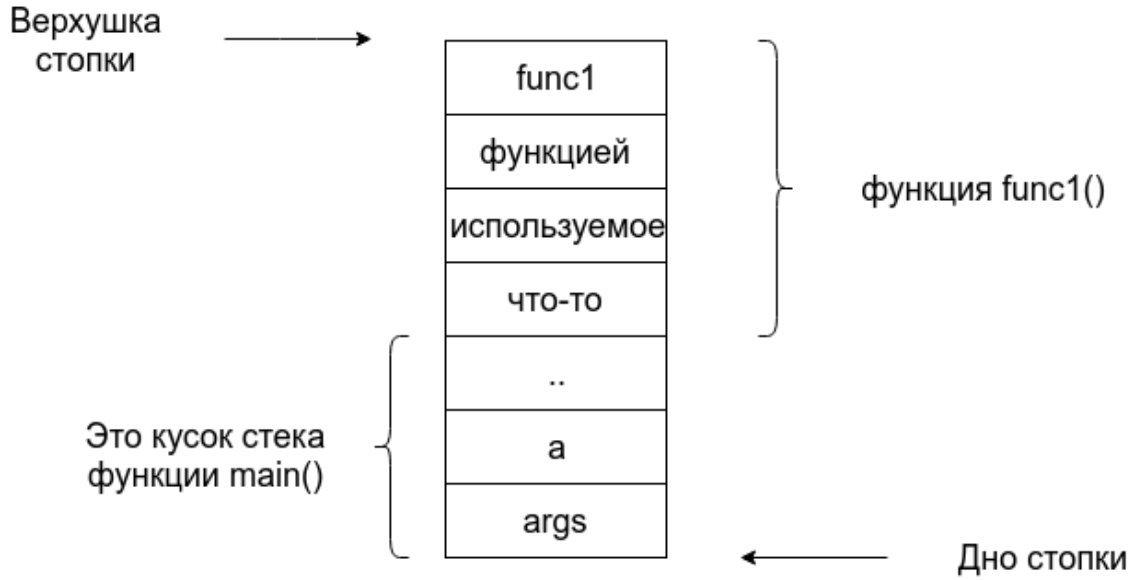
Давайте подробнее остановимся на моменте с очищением места в стеке.

Простой код:

```
public class App {
   public static void main(String[] args){
       int a = 3;
       int b = 4;
       int c = func1(a, b);
       c++;
       d = func2(c, a);
   }
    // something methods
}

```

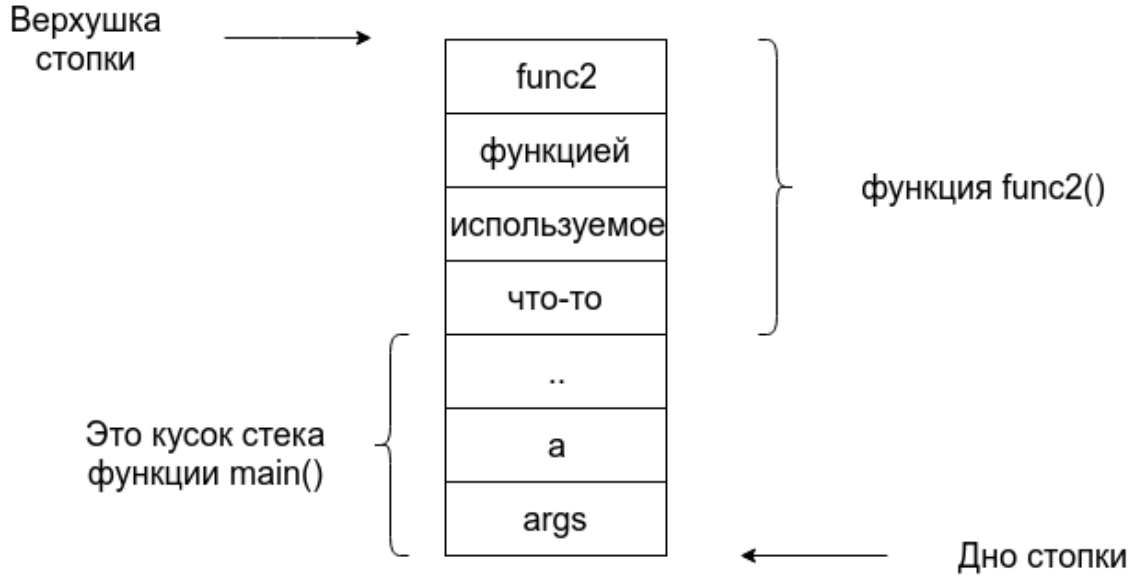
Допустим, что программа сейчас исполняет `func1` внутри метода `main` .

Стек будет выглядеть следующим образом:



Очень важно, понимать, что в этот момент времени в стеке нет ничего, что имеет отношение к функции `func2`.

В тот момент, когда мы войдем в `func2` часть стека используемая под `func1` будет стёрта, а на её месте будет `func2`



Итак, стеке хранятся данные, относящиеся к контексту функций, которые на этот момент времени выполняются. К таким данным относятся локальные переменные функции (то, что объявлено в её теле), аргументы функции, адрес возврата и возможно возвращаемое значение.

Наконец-то мы закончили со стеком, а теперь осталась куча. Что же это такое и как она работает?

Ссылочные типы данных и зачем нужна куча в Java?

Ранее в аргументах функции мы использовали простые типы данных, а что если нам необходимо использовать некоторые структуры?

Например, мы определили, что есть структура `человек` и она состоит из строки, описывающей имя человека и числа, описывающего его возраст. Получается, что когда нам нужно передать в функцию информацию о каком-то конкретном человеке (одна переменная), то нам нужно скопировать уже 2 значения. Вообще это похоже на то, как в примере выше копировались значения переменных a и b в аргументы функции.

Но иногда возникает такая ситуация, когда необходимо копировать достаточно много данных и если делать это достаточно часто, то будет много накладных расходов как по времени, так и по используемой памяти на хранение множества копий.

А что если функции передавать не само значение, а адрес, где оно лежит? К сожалению или к счастью, стек не предоставляет операции, где мы можем получить доступ к определенному месту в нем. К тому же, мы позволим какой-то функции `func1` влезть в данные функции `func2` , что не есть хорошо.

Также, если мы будем хранить нашу структуру в стеке, и создать экземпляр нашей структуры (класса) во внутренней функции, то передать ссылка на это место наверх к праотцам будет невозможно, так как в момент выхода, данные этой функции будут уничтожены (*см. главу Преимущество стека и области видимости*)

Таким образом, использование стека не предвещает ничего хорошего.

Тогда нам нужно еще одно хранилище данных, не имеющее вышеописанных минусов.

Для решения этих проблем было предложено сделать отдельную область памяти и назвать её **кучей (heap)**. Куча будет хранить какие-то долгоживущие объекты. Например нашу информацию о людях или о котиках.

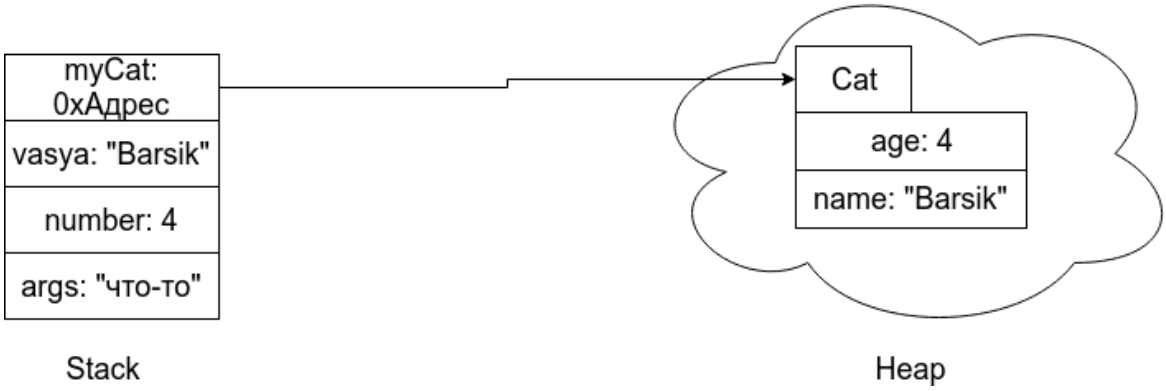
Теперь давайте представим, что у нас есть такой код на Java:

```
class Cat {
   String name;
   int age;
}

class App {
   public static void main(String[] args) {
      int number = 4;
      String vasya = "Барсик"; // Внешность обманчива :)
      Cat myCat = new Cat(vasya, number);
   }
}

```

В этом коде мы объявили некоторый класс (структуру), которая описывает кота и содержит информацию о его возрасте и имени. В функции `main` мы создали несколько переменных, в том числе создали нового кота.

Исходя из всего вышеописанного, не трудно догадаться, что созданный нами объект `Cat` будет хранится в куче по какому-то адресу памяти. Собственно, этот адрес будет хранится как переменная `myCat`, иными словами `myCat` только указывает на место хранения объекта и не хранит его значения.

Выглядит это так:



Как видно на картинке, **стек хранит адрес объекта, который лежит в куче**. Теперь если нам нужно будет передать нашего кота в какую-то другую функцию, то мы просто скопируем его адрес в стек другой функции. Получается, что какие-то переменные хранят адрес, а какие-то само значение. По этой причине в Java типы данных переменных разделяют на два типа. **Ссылочные** типы данных и **примитивные**. Примитивные типы хранят само значение, а ссылочные - адрес на место в кучи, где лежит объект.

К слову, была популярная задачка про сравнение String. Я её слегка изменил, но суть не изменилась:

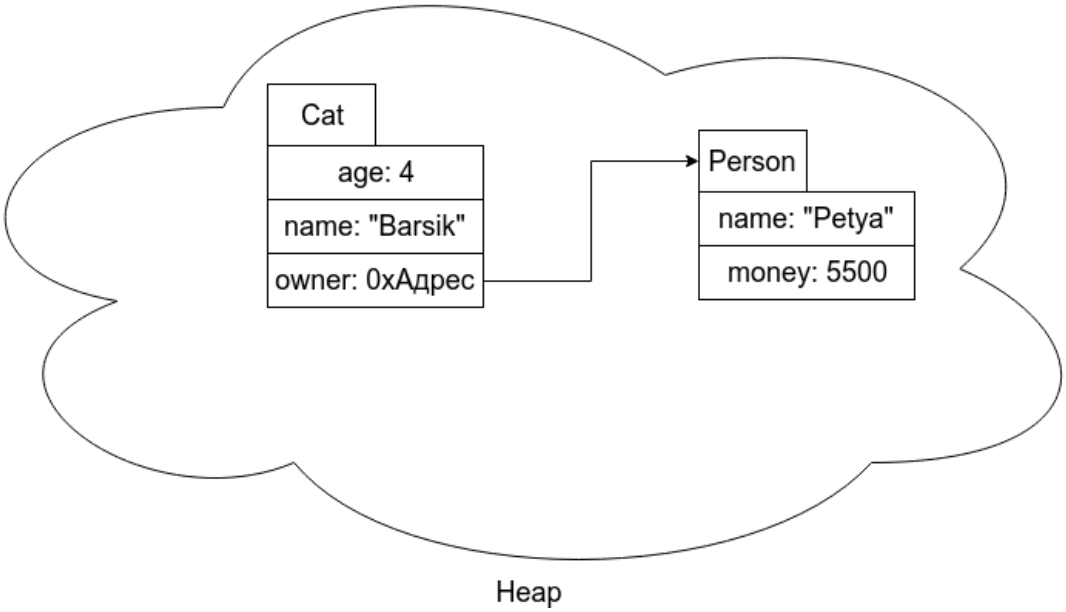
```
public class Task {
  public static void main(String ... args){
      String s1 = "123";
      func(s1);
  }

  static void func(String s1){
    String s2 = "123";
    System.out.println(s1 == s2);           // false
    System.out.println(s1 + " == " + s2);   // 123 == 123

  }
}

```

Раньше я уже упоминал про `String` - это не примитивный тип данных.

Стринги в Java это отдельная тема разговоров, где используются различные паттерны для их оптимизации (такие как String pool). Про них можно почитать в javarush.

Вернемся к куче.

Давайте представим, что у нашего кота появилась дополнительная информация о его владельце (отдельное поле в классе `Cat`), которая представлена информацией об имени и количестве денег у него:



Этот пример даёт понять важную вещь - адрес на объект не всегда хранится в стеке. В нашем примере владелец - часть информации о коте и если кот хранится в куче, то и ссылка на его владельца хранится там же. Но при этом так как `Person` - ссылочный тип данных, то он тоже хранится в куче, а у кота есть ссылка на него.

Ошибочно считать, что все данные о человеке будут находиться в том же кусочке где и данные кота. Также неверно считать, что если у нас переменная имеет примитивный тип данных, то она лежит в стеке. Возраст является частью кота, поэтому он находится там же где и информация о коте. Но при этом там лежит само значение 4, а не какой-то адрес, который указывает на место где записано 4.

Ластхит по стеку и куче

Из всего сказанного ранее следует, что в стеке хранится контекст исполняемых функций, а именно их локальные переменные, переданные в них аргументы, а также адрес возврата и возвращаемое значение. В зависимости от того какой тип имеют эти переменные (`ссылочный` или `примитивный`) в стеке могут лежать либо сами значения, либо адрес на место в куче. В куче же хранятся все объекты (которые являются ссылочными типами данных). Если объект содержит примитив, то внутри блока памяти отведенного под этот объект хранится само значение (в нашем примере 4), если же объект содержит ссылочный тип данных, то внутри него хранится адрес на другое место в куче, которое содержит информацию об этом объекте.

Парадигмы ООП

Давайте оставим байт-коды и прочее углубление в JVM и поднимемся на уровень выше - уровень абстракций.

В этой главе рассмотрим три основные парадигмы (абстракцию смотрите сами).

Инкапсуляция

Определение из википедии:

> Инкапсуляция — в информатике упаковка данных и функций в единый компонент.

Вроде бы верно, но это что-то слишком общее, что нельзя считать нормальным ответом.

Посмотрим, что пишут в методичке кафедры ВТ:

> Инкапсуляция - сокрытие данных внутри объекта и обеспечение доступа к ним с помощью общедоступных методов.

Это уже похоже на более внятный ответ. Разберем его получше.

Во-первых, следовало бы оговориться, что такие парадигмы ООП как инкапсуляция и полиморфизм - не относятся только к ООП, это лишь её парадигмы. Но здесь мы будем разбирать именно в контексте ООП.

Зачем нам нужно скрывать данные объекта?

Мы не должны давать другой системе (внешней) напрямую изменять свойства класса (состояние объекта). Если мы дадим любой системе изменять наши данные внутри объекта, то мы не сможем это корректно контролировать, что может привести к ошибке.

Поэтому мы должны предоставлять методы, которые позволят менять состояние нашего объекта

Приведу, очень простой пример, где инкапсуляция может помочь:

```
class MyClass{
    String val = "default";

    void printValLength(){
        System.out.println(val.length());
    }

}
public class Main {
    public static void main(String[] args) {
        MyClass a = new MyClass();
        a.printValLength();   // 7

        a.val = null;         // произвольно меняем значение
        a.printValLength();   // java.lang.NullPointerException
    }
}

```

Здесь, как вы видите, мы создаем класс с полем `val`, значение которой любая внешняя система может изменить. Как мы видим, в методе `printValLength` мы вызываем метод `legnth` , чтобы узнать длину строки. Но мы не можем знать, что значение `val` не равен `null` и поэтому, когда мы выполняем эту функцию, то есть шанс получить `NPE(NullPointerException)` .

```
class MyClass{
    private String val = "default";

```

```
    void printValLength(){
        System.out.println(val.length());
    }

    public void setVal(String val){
        if (val == null) this.val = "";
        else this.val = val;
    }

}
public class Main {
    public static void main(String[] args) {
        MyClass a = new MyClass();
        a.printValLength();   // 7

        a.setVal(null);
        a.printValLength();   // 0
  }
}

```

Здесь мы добавили проверку на `null` и можем гарантировать, что `val` не будет равен `null`.

Как же мы этого добились?

Во-первых, использовали модификатор доступа `private` тем самым ограничив область видимости нашего поля, тем самым не давая обращаться к этому полю за пределами тела класса.

Во-вторых, мы создали сеттер - `setVal`. По конвенции Java, следует называть сеттеры так:

`set<имя-поля>`, как сделали мы (в camelCase). Таким образом, другим программистам использующим ваш код или библиотеку, станет легче ориентироваться и он сразу будет знать как называется переменная, значение которой он изменяет.

По сути, мы сделали метод, который изменяет внутреннее состояние объекта. Здесь важно то, что объект сам изменяет своё состояние (совокупность свойств). То есть, изменяя переменную через такую функцию (сеттер), мы можем быть уверены, что никакой ошибки не будет (как например, NPE).

Разумеется, это не всегда так, бывают и ошибка, так как все мы люди, но всегда старайтесь делать так, чтобы изменение переменных через сеттер гарантировало безопасность.

К слову, есть обратное действие сеттеру - геттер, когда мы хотим не изменить значение поля, а получить его значение. Ведь, мы все равно не можем обратится к `private` переменной. Поэтому необходимо будет создать и метод, который возвращает значение поля:

```
class MyClass{
    private String val = "default";

```

```
    void printValLength(){
        System.out.println(val.length());
    }

    public void setVal(String val){
        if (val == null) this.val = "";
        else this.val = val;
    }

    public String getVal(){
        return this.val;
    }

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

        MyClass a = new MyClass();
        a.printValLength();   // 7
        System.out.println(a.getVal()); // default

        a.setVal(null);;
        a.printValLength();   // 0
  }
}

```

Модификаторы доступа

> Будьте бдительны! (с) Цопа

Есть 4 вида доступа ко внутренним свойствам класса, которые мы рассмотрели недавно:

- `package-private(package-visible, default)` - модификатор доступа по умолчанию. Его нельзя прописать вручную, так как в этом нет необходимости - можно просто не указывать модификатор доступа. Ограничивает область видимости пакетом класса, внутри которого он объявлен.
- `public` - неограниченная область видимости. Метод, поле или класс объявленный с этим модификатором можно использовать в любом месте проекта, если класс в котором он объявлен, также является публичным.
- `private` - ограничение области видимости в рамках тела класса.
- `protected` - ограничение области видимости внутри пакета и потомками, наследующих от класса, в котором он объявлен

Тут нужно быть очень внимательным, так как даже, если ваш метод объявлен как `public`, а класс в котором он находится `private`, то вы все равно не сможете получить к нему доступ.

Например,

```java
class Point {
    private class A {
      public void foo(){
        System.out.println("VT: krya krya!");
      }
    }
}
```

Мы не сможем получить доступ к методу `foo()` так как область его видимости ограничена не его модификатором, а модификатором класса, в котором он находится.

Также вас могут смутить модификаторы доступа в конструкторах:

```java
class Point {

    public Point(){}

    protected Point(int a){}

    Point(int a, int b){}

    private Point(int a, String s){}
}
```

Через области видимости конструкторов мы можем управлять тем, где может быть инициализирован наш класс.

Если с первыми тремя еще как-то понятно, но зачем нам модификатор доступа `private`?

Во-первых, её можно использовать через конструкцию `this()` для вызова конструкторов.

Во-вторых, если у нас все конструкторы будут `private`, то создать экземпляр класса можно будет только внутри его тела. Это может понадобится для реализации такого паттерна как `Singleton`

Наследование

Википедия:

> **Наследование** — концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения

Тут следует заметить, что под абстрактным типом подразумевается класс.

А вот более приближенное к Java определение кафедры ВТ:

> Наследование или расширение - приобретение одним классом (подклассом) свойств другого класса (суперкласса)

На самом деле, преимущества использования наследования очень полезны с использованием полиморфизма, но этому мы вернемся позже, а сейчас попробуем разобрать функционал, который дает нам наследование без учета полиморфизма.

Во-первых, это возможность наследования методов и даже полей. Допустим, у вас есть достаточно большой класс, содержащий, скажем, 12 методов. И вы должны создать класс, который имеет те же 12 методов, но и еще две дополнительные.

Первый вариант - это переписать весь тот большой класс в свой и добавить туда эти два метода. Казалось бы, почему бы и нет. Так, вы создали еще 5 классов, немного отличающихся друг от друга дополнительными методами. Но вдруг в один прекрасный день обнаружился баг в одном из методов, который присутствует в каждом из 7 классов, которые вы создавали. Вам сильно повезло, если вы записали в блокнот какие классы имеют вот этот одинаковый по реализации метод.

Второй вариант - это использование наследования. К счастью, наследование дает нам возможность избежать проблем первого варианта. Нам нужно просто наследоваться от нужного класса (который имеет 12 методов) и в своем классе определить лишь те методы, которых нет в суперклассе (класс от которого мы наследуемся).

Рассмотрим простой пример реализации наследования. Пример `inheritance/example1/InheritanceExample.java`

Определим наш класс человека

```
class Person {
    private String name;
    private int age;

    public void doSomething(){
        System.out.println("I'm individual person");
    }

    // геттеры / сеттеры
}

```

И давайте создадим босса:

```
class Boss extends Person {
    public void doSomethingLikeABoss(){
        System.out.println("I'm a boss!");
    }
}

```

Попробуем сделать какие-нибудь манипуляции:

```
Person person = new Person();

person.doSomething();       // I'm individual person
```

```
person.setAge(18);
person.setName("Lilith");

System.out.println(person.getName() + " " + person.getAge()); // Lilith 18

Boss boss = new Boss();

boss.setAge(30);        // методы наследованные от суперкласса (Person)
boss.setName("Boris");
System.out.println(boss.getName() + " " + boss.getAge());   // Boris 30

boss.doSomething();             // I'm individual person
boss.doSomethingLikeABoss();    // I'm boss!

```

Если посмотреть на класс `Boss`, то мы можем видеть только один метод, но по факту можем использовать все доступные методы из `Person`.

Что значит доступные? Так или иначе, модификаторы доступа к методам или полям при наследовании также остаются. При этом `private` методы или поля мы не сможем использовать даже в классе потомке (в нашем случае `Boss`).

> Задание: Попробуйте объявить поле age `protected` и обратится к ней напрямую из потомка.

И есть две очень важные вещи, касаемо наследования. Их мы и разберем.

Давайте, добавим конструктор в класс `Person` , потому что задавать их вручную - неудобно.

Пример `inheritance/example2/InheritanceExample2.java`

```
class Person {
    private String name;
    private int age;

    public Person(String name, int age){
        this.name = name;
        this.age = age;
    }

    public void doSomething(){
        System.out.println("I'm individual person");
    }

    // геттеры и сеттеры
}

```

Как только мы объявили конструктор в родительском классе, то должны объявить его и в классе потомке, если нету конструктора по умолчанию.

Разберем это подробнее. Как мы ранее говорили (см. главу Конструкторы) у всех классов есть конструктор по умолчанию. Этот конструктор пустой. При наследовании важно понимать, что если мы объявим какой-то непустой конструктор в суперклассе, тогда конструктор по умолчанию исчезнет, а следовательно его не будет и в классе потомке. Поэтому мы должны создать этот же конструктор и в классе потомке и прописать как он будет работать.

К счастью, нам не придется копировать код который находится в предке, а можем использовать `super`. Это переменная позволяет обращаться к конструктор класса, от которого мы наследуемся.

```
class Boss extends Person {
    public Boss(String name, int age){
        super(name, age);   // обращаемся к конструктору Person
    }

    public void doSomethingLikeABoss(){
        System.out.println("I'm a boss!");
    }

}

```

Переопределение методов

Еще одной важной вещью для использования наследования является переопределение.

Вы можете использовать переопределение, если хотите поменять поведение метода. Например, мы можем переопределить метод `doSomething` у `Boss`:

```
class Boss extends Person {
    public Boss(String name, int age){
        super(name, age);
    }
    public void doSomething(){
        System.out.println("I'm individual boss!");
    }

    public void doSomethingLikeABoss(){
        System.out.println("I'm a boss!");
    }
}

```

И увидеть, что выполнится метод, который объявлен в `Boss`, а не в `Person`:

```
Person person = new Person("Lilith", 18);
person.doSomething();       // I'm individual person

Boss boss = new Boss("Boris", 30);
boss.doSomething();         // I'm individual boss!

```

> Умение находить информацию в интернете - очень важная часть работы. Когда вы можете получить все из одного места, то нельзя развить навыки поиска решений в интернете.
>
> Поэтому, сейчас возьмите и узнайте, что такое @Override. Не страшно, если вы не знаете аннотации, о них мы еще поговорим, но использовать @Override вы уже можете.

Наследуются ли конструкторы?

- Нет, не наследуются!

Итак, почему же конструкторы не наследуются? Дело в том, что если мы наследуем все конструкторы родителя автоматически, то не можем гарантировать, что какая-то переменная не инициализируется неправильно.

Например, мы можем добавить какой-то метод, который высчитывает длину строки имени, а конструктор суперкласса может инициировать эту переменную как `null`. В свою очередь, мы можем не заметить этого и получить ошибки NPE.

Поэтому, если мы хотим иметь те же конструкторы, что и родитель (суперкласс), то мы должны явно их объявить, как мы это сделали с примером `Boss`.

> Забавный факт: Если вы знаете класс Object, от которого наследуются все классы, то представьте, что было бы, если бы все конструкторы наследовались по умолчанию :)

О работе super и переопределения конструкторов

Здесь мы коснемся важной темой переопределения конструкторов.

Сначала, требуется понимать, что когда вы вызываете конструктор потомка, то сначала вызывается конструктор предка.

Это легко можно проверить не залезая в байт-код. Пример `example3/ConstructorInhExample.java`:

```
1  class A{
2      public A(){
3          System.out.println("A constructor");
4      }
5  }
6  class B extends A {
7      public B(){
8          System.out.println("B constructor");
9      }
10 }
11
12 public class ConstructorInhExample {
13     public static void main(String ... args){
14         B b = new B();
15
16         // Output:
17         // A constructor
18         // B constructor
19     }
20 }
21
```

Как мы видим, в конструкторе B мы не вызывали `super()`, но все же он вызвался. То есть всегда по умолчанию вызывается пустой конструктор предка, если мы сами его не вызываем. Сделано это для того чтобы переменные из предка инициализировались правильно.

Проблемы могут начаться тогда, когда мы объявим свой конструктор, тем самым убрав конструктор по умолчанию. Тогда конструктор из `B` попытается вызвать пустой конструктор по умолчанию, но ничего не найдет, так как его в предке нет.

Решить эту проблему, можно вызвав `super` вручную с какими-то аргументами, которая объявлена в предке. Например, мы сделали это в примере `inheritance/example2/InheritanceExample2.java`.

Добавление новых конструкторов

Если вы хотите добавить свой конструктор, то вызывайте `super` вручную в соответствии с тем, что объявлено в предке. Но если вы этого не сделаете, то ваш конструктор неявно (автоматически) добавит `super()`, заметьте с пустыми аргументами. В том случае, когда пустой конструктор в предке отсутствует - вы получите ошибку на этапе компиляции.

Полиморфизм

Наконец, мы дошли до третьей парадигмы - полиморфизм.

Параметрический полиморфизм

Если коротко, то

> Параметрический полиморфизм - способность функции обрабатывать данные разных типов

Опять же представим себе ситуацию, когда нам нужно передавать в аргументы функции несколько разных типов. Например, возвращаясь к примеру с боссом и человеком - мы хотим создать метод, который на вход получает босса **или** человека, а потом выводит его имя.

Тут вы уже могли заметить, что босс по сути является потомком человека - содержит все методы, которые есть у человека. А значит, мы можем указать в функции тип человека и при этом передавать туда босса (потому что это его потомок).

И ведь, действительно, если босс является потомком человека, мы можем гарантировать, что у него есть все методы человека и ошибки вроде `No Such Method Error` или подобного не выйдет.

Пример `polymorph/PolymorphExample.java`

```
1  class Person {
2      private String name;
3      private int age;
4
5      public Person(String name, int age){
6          this.name = name;
7          this.age = age;
8      }
9      public Person(String name, int age, int i){
```

```
        this.name = name;
        this.age = age;
    }

    public void doSomething(){
        System.out.println("I'm individual person");
    }

    // геттеры\сеттеры
}

class Boss extends Person {
    public Boss(String name, int age){
        super(name, age);
    }
    public void doSomethingLikeABoss(){
        System.out.println("I'm a boss!");
    }

}

public class PolymorphExample {
    private static void myPolymorphMethod(Person person){
        System.out.println(person.getName());
    }
    public static void main(String ... args) {
        Person person = new Person("Alice", 21);
        Boss boss = new Boss("Juan", 38);

        myPolymorphMethod(person);      // Alice
        myPolymorphMethod(boss);        // Juan
    }
}

```

Полиморфизм подтипов

Это свойство позволяет обращаться с помощью единого интерфейса к классу и к любым его потомкам, также его называют полиморфизмом включения.

```
class A {
    public void doSome(){
        System.out.println("A doing something");
    }
}

class B extends A{
    @Override
    public void doSome(){
        System.out.println("B doing something");
    }
}


public class PolymorphExample2{
```

```
    public static void main(String ... args){
        A obj1 = new A();
        A obj2 = new B();   // полиморфизм in action!

        obj1.doSome();  // A doing something
        obj2.doSome();  // B doing something

        // point 1
    }
}

```

Здесь мы видим, что мы можем объявить тип `obj2` как тип `A`, но при этом создать для него экземпляр класса `B` . Такие операции называют **восходящим преобразованием** - его мы разберем потом.

> **Задание**: попробуйте добавить новый метод в класс B и вызвать его в этом же коде, после вызова двух методов (point 1). Объяснить это явление вам поможет восходящее преобразование

Слишком просто? Тогда давайте углубимся в полиморфизм, затрагивая темы динамического связывания, восходящего преобразования и поведения полиморфных методов при вызове из конструкторов.

Но если вы не любите усложнять себе жизнь или если вы казуал, то можете посмотреть статью на javarush, где вполне доступно, как для детей, рассказывают про полиморфизм с картинками.

Динамическое связывание

... печатает текст

Подробнее об инициализации

Блоки инициализации

Язык Java позволяет сгруппировать несколько действий по инициализации объектов `static` в специальной конструкции, называемой *статическим блоком*. А также для инициализации нестатических переменных (без `static`) каждого объекта просто блоки инициализации.

Посмотрим пример `examples/manual-2/block/example-1/BlockOfInitExample.java`

```
public class BlockOfInitExample{

    private int a = 0;
    private static int b = 0;


    {           // нестатический блок инициализации
```

```
        a = 10;
    }

    static {    // статический блок инициализации
        b = 10;
    }

    public static void main(String ... args){

        System.out.println("b: " + b);                          // 10

        System.out.println("a: " + new BlockOfInitExample().a); // 10
    }

}

```

Статический блок кода, как и остальная инициализация `static`, выполняется лишь один раз: при первом создании объекта этого класса или при первом обращении к статическим членам этого класса (даже если ни один объект класса не создается).

А вот нестатический блок инициализации выполняется каждый раз, когда создается объект. При этом выполняется, когда выполнены все статические блоки инициализации и **до** конструкторов.

Рассмотрим следующий код:

`example-2/BlockOfInitExample2.java`

```
public class BlockOfInitExample2 {

    private int a = 0;
    private static int b = 0;

    {
        System.out.println("non-static block");
        a = 10 + a;
        b = 60 + b;     // можем изменять статические поля
    }

    static {
        System.out.println("static block");
        b = 55;

        // мы не можем здесь определить переменную "a"
        // т.к. a - нестатическая переменная
        // a = 90;
    }

    private void printValues(){
        System.out.println("a: " + a);
        System.out.println("b: " + b);
    }

    public static void main(String ... args){

        new BlockOfInitExample2().printValues();    // #1
```

```
        // static block
        // non-static block
        // a: 10
        // b: 115       // 55 + 60

        new BlockOfInitExample2().printValues();    // #2
        // non-static block
        // a: 10
        // b: 175       // 115 + 60

        new BlockOfInitExample2().printValues();    // #3
        // non-static block
        // a: 10
        // b: 235       // 175 + 60
    }

}

```

Метод `printValues` выводит значения переменных `a` и `b`

Как мы можем видеть, статический блок инициализации срабатывает только одиножды - когда мы впервые создаем объект. Также можно заметить, что нестатический блок инициализации сработал позднее статического, но исполняется каждый раз, когда создается объект, несмотря на то, что статический блок инициализации находится ниже, чем он.

> **Задание**: напишите пример кода, где видно, что конструктор исполняется после исполнения нестатического блока инициализации.

Также следует обратить особое внимание на порядок вызовов конструкторов и блоков инициализации при наследовании:

```
class A {
    A(){
        System.out.println("constructor [A]");
    }
    static {
        System.out.println("static block of init [A]");
    }

    {
        System.out.println("block of init [A]");
    }
}

class B extends A {
    B(){
        System.out.println("constructor [B]");
    }
    static {
        System.out.println("static block of init [B]");
    }

```

```
    {
        System.out.println("block of init [B]");
    }
}

public class BlockOfInitExample3 {
    public static void main(String ... args){
        System.out.println("First B init:");
        new B();

        System.out.println("Second B init:");
        new B();
    }
}

// Output:

// First B init:
// static block of init [A]
// static block of init [B]
// block of init [A]
// constructor [A]
// block of init [B]
// constructor [B]

// Second B init:
// block of init [A]
// constructor [A]
// block of init [B]
// constructor [B]

```

> **Задание**: разберите, почему на выводе мы получаем такой результат.
>
> Совет: см. главу Конструкторы

Такой синтаксис необходим для поддержки *анонимных внутренних классов* , но он также гарантирует, что некоторые операции будут выполнены независимо от того, какой именно конструктор был вызван в программе.

Что за `new`?

Вы уже могли много раз использовать оператор `new`. Как вы уже поняли - он нужен для создания объекта.

Оператор `new` :

- Выделяет место для объекта
- Вызывает конструктор объекта, который инициализирует объект
- Возвращает ссылку на объект в памяти

Что за `.new`?

Иногда в программе требуется приказать объекту создать объект одного из его внутренних классов. Для этого в выражение `new` включается ссылка на другой объект внешнего класса с синтаксисом `.new`

К примеру, если мы имеем внешний класс `A`, который имеет внутренний класс `B` (класс, который объявлен внутри другого класса) , то мы можем создать объект класса `B` , обратившись к объекту класса `A` , таким образом: `objA.new B()` , как-будто используем обычный `new`

Пример `manual-2/new/example-1/InnerNewExample.java`

```
public class InnerNewExample {
    private class Inner {};
    public static void main (String ... args){
        InnerNewExample obj = new InnerNewExample();
        Inner innerObj = obj.new Inner();
    }
}

```

Инициализация массивов

Переменные массивов не содержат значения, а только ссылку на сам массив (логично, так как они создаются с помощью оператора `new`, либо память выделяется неявно компилятором)

Посмотрим пример `manual-2/array/example-1/ArraysExample.java`

```
public class ArraysExample {
    public static void main(String ... args){
        int[] a1 = {1, 2, 3, 4, 5}; // выделение памяти (new) производится компилятором
        int a2[];

        a2 = a1;    // присваиваем ссылку, а не сам массив

        for (int i = 0; i < a2.length; i++){
            a2[i] = a2[i] + 1;
        }

        for (int i = 0; i < a1.length; i++){
            System.out.printf("a1[%d] = " + a1[i] + "\n", i);
        }
    }
}

// Output:
// a1[0] = 2
// a1[1] = 3
// a1[2] = 4
// a1[3] = 5
// a1[4] = 6
```

В данном случае `a2 = a1` вы, на самом деле, копируете ссылку, что продемонстрировано при выводе значений массива `a1`

Модификатор `final`

`final` для полей и переменных

Как правило, модификатор `final`, говорит о том, что объект не должен изменяться.

Это может понадобится, когда вы хотите объявить переменную, значение, которой не должно меняться. И тут речь идет не о внутренних данных вашей переменной, а именно сама переменная, представленная в виде ссылки.

Рассмотрим простой пример:

```
final int A = 3;
A = 4;  // ошибка компиляции
```

Вроде бы, все тривиально. Но как насчет ссылочных типов данных?

Создадим свой класс (для упрощения без геттеров и сеттеров):

```
class Point{
  int a;
  int b;
  Point(int a, int b){
    this.a = a;
    this.b = b;
  }
}
```

Пробуем:

```
final Point point = new Point(1, 2);
point = new Point(1,2); // ошибка компиляции
```

Отлично, действительно, `final` гарантирует нам, что ссылка на объект не изменится. Но это касается только ссылки на объект, но не сами значения объекта. Ведь, `point` - это только ссылка на наш объект. Поэтому, мы вполне можем сделать и такое:

```
final Point point = new Point(1, 2);
System.out.println(point.a + " " + point.b);    // 1 2
point.a = 55;
point.b = 111;
System.out.println(point.a + " " + point.b);    // 55 111
```

Выглядит логично, так как мы объявили `final` лишь ссылку на объект.

Чтобы значения нашего `Point` также не менялись, нам следует объявить их `final`

```
class Point{
  final int a;
  final int b;
  Point(int a, int b){
    this.a = a;
    this.b = b;
  }
}
```

В этом случае, важно понимать, что нам нужно инициализировать `final`-переменные до создания готового экземпляра. Иначе говоря, когда `new` возвращает ссылку на наш объект, то все `final`-переменные должны быть проинициализированы.

За этим проследит компилятор, но за его реализацию ответственен сам программист. В Java инициализировать final переменную можно несколькими способами:

- При объявлении
- В блоке инициализации
- В конструкторе

Также, следует заметить, что мы можем проинициализировать нашу переменную лишь один раз, то есть случай, когда мы сделаем инициализацию два раза должен быть исключен.

```
class Point{
  final int a;
  final int b;
  final static int C;
  final static int D = 4;

  static {  // статический блок инициализации
    C = 3;
  }

  {         // блок инициализации
    b = 2;
    // D = 4;
  }

  Point(int a){
    this.a = a;
  }

  Point(){
    a = 1;
  }
}
```

Что мы можем уяснить с этого примера?

Во-первых, мы не можем проинициализировать `static`-переменную в не статическом блоке. В следствие того, что нестатический блок инициализации исполнится только при создании экземпляра.

Во-вторых, мы можем по-разному и в нескольких местах написать инициализацию `final`-переменной (в примере это поле `a`). Но при этом мы должны гарантировать, что случаи инициализации этой переменной взаимоисключающие.

Конкретно в этом примере, мы инициализируем его в одном из конструкторов. Очевидно, что два конструктора здесь не вызовутся. Ради проверки, вы можете поставить в один из конструкторов вызов другого через `this()` или `this(int a)`. Тогда компилятор уже сообщит вам об ошибке.

Примечательным также является и то, что у `final`-переменных нет значения по умолчанию, поэтому выполнив такой код, мы получим ошибку:

```
class Point{
  private final static int A;
  static {
    System.out.println(A);  // variable a might not have been initialized
    A = 7;
  }
}
```

И напоследок, чтобы собрать всю информацию воедино, посмотрим пример еще одной ошибки:

```
class Point {
    static final int VAR;

    static {
        Point.VAR = 10; // обращение через класс
    }
}
```

В этом примере интересна не столько ошибка, сколько и какая из ошибок вызовется первым: переменная не инициализирована или изменение `final`-переменной.

Оказывается, что компилятор сначала будет ругаться на то, что мы попытаемся изменить `final`-переменную и только потом на то, что она не проинициализирована. Это ожидаемо, так как компилятор может предположить, что переменная будет проинициализирована где-то еще в другом блоке инициализации.

Именование `final`-полей и переменных

Соблюдать code-style какого-то языка всегда считается хорошим тоном, а если вы хотите, чтобы другой программист прочитал ваш код, то это даже необходимость. Такой code-style также есть отдельно для константных переменных.

Все константы должно быть написаны в CONSTANT_CASE

```
// Constants
static final int NUMBER = 5;
static final long NUMBER = 10L;
static final ImmutableList<String> NAMES = ImmutableList.of("Ed", "Ann");
static final SomeMutableType[] EMPTY_ARRAY = {};
```

Но раньше вы могли видеть, что при объявлении экземпляра `Point` мы назвали её как обычную переменную:

```
class Point{
  int a;
  int b;
  Point(int a, int b){
    this.a = a;
    this.b = b;
  }
}

final Point point = new Point(1, 2);
```

Дело в том, что любой экземпляр класса `Point` не является константой, так как можно изменить её внутреннее состояние. К примеру, мы можем изменить значения полей `a` и `b`

Если мы внутреннее состояние нашего экземпляра не могло бы меняться, то мы бы именовали её как константу:

```
class Point{
  final int a;
  final int b;
  Point(int a, int b){
    this.a = a;
    this.b = b;
  }
}

final Point POINT = new Point(1, 2);    // constant
```

Тоже самое относится, например, к непустым массивам (мы можем по индексу менять их значение), **не статическим** `final`-переменным (можно менять их значение при создании класса), стандартным коллекциям:

```
// Not constants
static String nonFinal = "non-final";
final String nonStatic = "non-static";
static final Set<String> mutableCollection = new HashSet<String>();
static final String[] nonEmptyArray = {"these", "can", "change"};
```

`final` для методов

Модификатор `final` применим и для методов.

Давайте подумаем, как он может применяться в контексте методов. Раз уж мы говорим `final`, то следовательно, метод не должен меняться. А когда он может меняться? - При переопределении в случае наследования. Значит, это метод, который нельзя переопределить.

Действительно, все звучит очень логично:

```
class Point {
    final void foo(){
        // something
    }
}

class PointChild extends Point {
    void foo(){ // can't override

    }
}
```

`final` для классов

Оказывается, что `final` может использоваться и для классов.

```
final class Point {
    final void foo(){
      // something
    }
}

// error: can't inherit
class PointChild extends Point{

}
```

Как видно из примера, мы здесь не можем наследоваться от класса, то есть опять же не можем изменить этот класс. Логика осталась прежней.

Приложение А. Использованная литература

Arhipov Blogspot - Java Bytecode Fundamentals

jrebel.com - Mastering Java Bytecode at the Core of the JVM

dzone.com - Introduction to Java Bytecode

shipilev.net - JVM Anatomy Quark #8: Local Variable Reachability

Jamesdbloom Blog - JVM Internals

Tim Lindholm, Frank Yellin - JVM Specification Java SE8 Edition

Отдельная благодарность статье на Tune-IT, именно она сподвигла меня на идею написать подробнее про стек и кучу:

Alexander Yarkeev - Стек и куча для чайников

Коллекции

Коллекции в Java - это множество различных структур данных, включая 'динамический массив', 'стек', 'очередь', 'двунаправленная очередь', 'сет' и так далее.

Но перед тем как приступить к их изучению, я предлагаю ознакомится с одной интересной возможностью создания параметризованных типов, именуемых как `Generics`

Generics

Жизнь до Java 5

Допустим, мы на этапе написания кода не знаем какой именно конкретный тип нам нужно возвращать. Предположим, что это станет известно, например, только тогда, когда пользователь совершит ввод.

Как же тогда нам писать метод, который был бы универсальным под любой ввод пользователя?

Мы можем возвратить не конкретный тип объекта, а какую-то его абстракцию (суперкласс)

```
public SomethingInterface getFoo(args){ ... }
public SomethingClass getFoo(args){ ... }
```

Будем считать, что `getFoo()` возвращает любой предок имплементирующий или наследующий от `SomethingInterface` или `SomethingClass` , соответственно.

Согласитесь, это накладывает определенные ограничения и делает наш код более небезопасным.

Во-первых, чтобы сконструировать `getFoo()` таким образом, чтобы он работал с любым пользовательским классом невозможно (если это не `Object`), так как он может возвращать только потомки возвращаемого типа.

Во-вторых, получая от метода какую-то абстракцию, мы не сможем использовать его специфичные методы реализованные в имплементациях или потомках. Чтобы использовать их, нам нужно будет использовать `casting`

Решая первую проблему, мы несомненно будем понижать безопасность использования этого метода, так как нам нужно решать и вторую проблему с `casting`

Для этого приведем простой пример:

```
public Object getFoo(){return ...}

// ...

Object obj = getFoo();

if (obj instanceof List){
    System.out.println("It is list!");
} else if (obj instanceof String){
    System.out.println("It is String");
} else {
    System.out.println("We need more IF!");
}
```

```
...
```

Как видно из примера, либо мы сильно увеличиваем наш код в длину, проверяя возвращаемый тип, либо получим какой-нибудь `ClassCastException`, когда попытаемся сделать преобразование несоответствующих типов:

```
Object obj = getFoo();          // returns List
String answer = (String) obj;   // error: ClassCastException
```

И пришел Java 5

> JSR-000014: Add Generic Types to the Java Programming Language

Начиная с 5-ой версии Java появилась возможность использовать `Generic Types`

```
class Point {
    <T> void foo(T arg){
      // somehting
    }
}

// ...

Point p = new Point();
p.<String>foo("arg with String type");
p.<Integer>foo(12);

p.<Integer>foo("invalid param"); // compilation error!
```

То есть, условно вместо `T` будет наш тип указанный внутри `<Type>`

Также параметризацию можно указать перед телом класса:

```
class Point <T>{
    void foo(T arg){
      // somehting
    }
}

// ...

Point p = new Point<String>();
p.foo("arg with String type");

p.foo(new AnotherClass());  // compilation error!
```

Одним из немаловажных преимуществ использования параметризованных типов является и то, что нам нет необходимости использовать преобразование типов. Компилятор уже может гарантировать какой тип данных там есть.

На самом деле, этих знаний вполне достаточно, если вы будете производить базовые операции с коллекциями, но если вы хотите сами создавать обобщенные методы, то вам следует понимать как обобщенные типы устроены внутри, ведь компилятор зачастую выполняет больше работы, чем мы о нем думаем.

Разница между Object и Generic type

Давайте посмотрим во что превращается наш код после компиляции.

Исходный код:

```
public class GClass<T> {
        public T getFoo(T someObject){
                return someObject;
        }

        public String getFoo(String someString){
                return "String";
        }
}
```

Байт-код после компиляции исходного:

```
  public GClass();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."
<init>":()V
         4: return
      LineNumberTable:
        line 1: 0

  public T getFoo(T);
    descriptor: (Ljava/lang/Object;)Ljava/lang/Object;
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=2, args_size=2
         0: aload_1
         1: areturn
      LineNumberTable:
        line 3: 0
    Signature: #12                          // (TT;)TT;

  public java.lang.String getFoo(java.lang.String);
    descriptor: (Ljava/lang/String;)Ljava/lang/String;
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=2, args_size=2
         0: ldc           #2                  // String String
         2: areturn
      LineNumberTable:
        line 7: 0
}
```

```
33 Signature: #14                          //
   <T:Ljava/lang/Object;>Ljava/lang/Object;
34 SourceFile: "GClass.java"
```

Мы объявили два метода `getFoo()`, но один из них имел тип `String`, а другой обобщенный.

Действительно, в байт-коде есть два наших метода, но у первого из них другой дескриптор, который содержит тип возвращаемого значения и тип его сигнатуры.

Если вам интересен, как устроен байт-код внутри JVM, то советую прочитать главу Hello World из байт-кода в конце книги в главе "Самым любознательным".

У дескриптора, есть определенные правила. В скобках пишется сигнатура метода (аргументы), а после него слитно пишется тип возвращаемого значения.

> (Тип-сигнатуры)Тип-возвращаемого-значения

В дескрипторе метода `public T getFoo(T)` мы видим `Object` как тип аргументов, и такой же `Object` как возвращаемый тип. Из-за чего это происходит?

Дело в том, что при компиляции, мы не можем знать какой именно тип там находится, что приводит к тому, чтобы использовать самый обобщенный тип, чтобы не вызывать конфликты. Разумеется, следует вопрос:

> "А почему именно так? Разве мы не могли сделать что-то на уровне полноправных сущностей?".

Чтобы понять почему было принято такое решение, необходимо уяснить, что обобщенные типы появились только в 5 Java, а значит до него было написано много библиотек и фреймворков без использования обобщенного программирования.

Разработчикам языка (особенно Java) необходимо было сохранять обратную совместимость (существующий код и файлы классов остаются действительными, а их смысл не изменился). Кроме того, есть такое понятие как поддержка миграционной совместимости, чтобы авторы библиотеки могли заниматься их обобщением в нужном темпе, а когда библиотека становится обобщенной - она не нарушает работоспособности кода и приложений, которые от неё зависят.

Итак, разработчики языка решили использовать **стирание типов**, что мы можем видеть на предыдущем примере. Все обобщенные типы по сути своей являются в байт-коде типом `Object`

Разумеется, компилятор не просто берет и стирает все обобщенные типы до `Object`, а создает определенные метки для JVM, чтобы тот тоже понял, что на самом деле, это обобщенный тип.

Поэтому в конце метода вы можете видеть такую запись:

```
1    Signature: #12                          // (TT;)TT;
```

Стирание типов является не самым приятным решением, который накладывает ряд ограничений. Например, если мы в предыдущем примере объявили два метода `getFoo()` с аргументами `String` и обобщенным, то что будет, если мы вместо `String` используем `Object`?

Как и следовало ожидать мы получим ошибку:

```
public class GClass<T> {
    public T getFoo(T someObject){
        return someObject;
    }

    public Object getFoo(Object someString){
        return "String";
    }
}
```

Ведь, как мы уже видели - обобщенные типы стираются до `Object`, что приводит к конфликту.

Но это малое, чем мы может расплатится за стирание. Больнее всего то, что мы не можем получить данные о нашем типе. Если они стираются до `Object` о каких специфичных методах может идти речь?

Таким образом, любые операции требующие знания точного типа во время компиляции работать не будут. В частности, создание массива или создание экземпляра.

И возникает вопрос, а есть ли вообще отличие от `Object`? Да, разумеется. Во-первых, нам не нужно производить приведение типов. Также, если мы зададим обобщенный тип, то условно параметризуем типы, как было приведено в предыдущей главе.

Но существенное отличие от `Object` - это то, что мы можем ограничить стирание типов. То есть, сказать компилятору не стирать тип до `Object`, а до чего-то определенного. Очевидно, что это накладывает и ограничения на пользователя, который будет использовать ограниченный обобщенный тип, но это даем нам возможность использовать специфичные методы определенного класса или интерфейса.

Так как все в Java наследуется от `Object, то вполне законно будет написать такое:

```
public class GClass<T extends Object> {
        public T getFoo(T someObject){
                return someObject;
        }
}
```

Он будет эквивалентен обобщенному методу `getFoo` из первой записи без `...extends Object`.

Но к примеру, мы можем записать:

```
public class GClass<T extends String> {
        public T getFoo(T someObject){
                return someObject;
        }
}
```

И получить в методе `getFoo` специфичные методы, такие как `matches` или `substring`

Не стоит также забывать, что при объявлении обобщенного типа при создании класса, мы также имеем ограничение указать там тип который либо `String`, либо является его потомком.

Печатает текст...

Самым любознательным

В этой секции будут темы, которые могут быть за гранью контекста Java или связаны с низкоуровневым программированием

Hello World из байт-кода

Для начала создадим простенькую программу:

```
    public class Main {
        public static void main(String ... args) {
            System.out.println("Hello World");
        }
    }
```

Скомпилируем её командой `javac Main.java` и собственно сделаем дизассемблинг

```
    javap -c -v Main
```

Main.class

```
Classfile /C:/Users/Arthur/playground/java/jvm/Main.class
  Last modified 26.10.2019; size 413 bytes
  MD5 checksum 6449121a3bb611fee394e4f322401ee1
  Compiled from "Main.java"
public class Main
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #6.#15         // java/lang/Object."<init>":()V
   #2 = Fieldref           #16.#17        //
java/lang/System.out:Ljava/io/PrintStream;
   #3 = String             #18            // Hello World
   #4 = Methodref          #19.#20 // java/io/PrintStream.println:
(Ljava/lang/String;)V
   #5 = Class              #21            // Main
   #6 = Class              #22            // java/lang/Object
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               main
```

```
  #12 = Utf8               ([Ljava/lang/String;)V
  #13 = Utf8               SourceFile
  #14 = Utf8               Main.java
  #15 = NameAndType        #7:#8          // "<init>":()V
  #16 = Class              #23            // java/lang/System
  #17 = NameAndType        #24:#25        // out:Ljava/io/PrintStream;
  #18 = Utf8               Hello World
  #19 = Class              #26            // java/io/PrintStream
  #20 = NameAndType        #27:#28        // println:(Ljava/lang/String;)V
  #21 = Utf8               Main
  #22 = Utf8               java/lang/Object
  #23 = Utf8               java/lang/System
  #24 = Utf8               out
  #25 = Utf8               Ljava/io/PrintStream;
  #26 = Utf8               java/io/PrintStream
  #27 = Utf8               println
  #28 = Utf8               (Ljava/lang/String;)V
{
  public Main();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."
<init>":()V
         4: return
      LineNumberTable:
        line 1: 0

  public static void main(java.lang.String...);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC, ACC_VARARGS
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #2        // Field
java/lang/System.out:Ljava/io/PrintStream;
         3: ldc           #3        // String Hello World
         5: invokevirtual #4// Method java/io/PrintStream.println:
(Ljava/lang/String;)V
         8: return
      LineNumberTable:
        line 4: 0
        line 5: 8
}
SourceFile: "Main.java"

```

Это просто представление байт-кода, которое человеку видеть легче, чем оригинальный байт-код, но сам он выглядит иначе:

```
    cafe babe 0000 0034 001d 0a00 0600 0f09
    0010 0011 0800 120a 0013 0014 0700 1507
    0016 0100 063c 696e 6974 3e01 0003 2829
    5601 0004 436f 6465 0100 0f4c 696e 654e

```

```
    756d 6265 7254 6162 6c65 0100 046d 6169
    6e01 0016 285b 4c6a 6176 612f 6c61 6e67
    2f53 7472 696e 673b 2956 0100 0a53 6f75
    7263 6546 696c 6501 0009 4d61 696e 2e6a
    6176 610c 0007 0008 0700 170c 0018 0019
    0100 0b48 656c 6c6f 2057 6f72 6c64 0700
    1a0c 001b 001c 0100 044d 6169 6e01 0010
    6a61 7661 2f6c 616e 672f 4f62 6a65 6374
    0100 106a 6176 612f 6c61 6e67 2f53 7973
    7465 6d01 0003 6f75 7401 0015 4c6a 6176
    612f 696f 2f50 7269 6e74 5374 7265 616d
    3b01 0013 6a61 7661 2f69 6f2f 5072 696e
    7453 7472 6561 6d01 0007 7072 696e 746c
    6e01 0015 284c 6a61 7661 2f6c 616e 672f
    5374 7269 6e67 3b29 5600 2100 0500 0600
    0000 0000 0200 0100 0700 0800 0100 0900
    0000 1d00 0100 0100 0000 052a b700 01b1
    0000 0001 000a 0000 0006 0001 0000 0001
    0089 000b 000c 0001 0009 0000 0025 0002
    0001 0000 0009 b200 0212 03b6 0004 b100
    0000 0100 0a00 0000 0a00 0200 0000 0400
    0800 0500 0100 0d00 0000 0200 0e
```

С этим кодом мы и будем работать.

Но для начала нам нужно его форматировать, чтобы не путаться что где находится, а байт-код, на самом деле, имеет вполне жесткую структуру:

```
    ClassFile {
        u4             magic;
        u2             minor_version;
        u2             major_version;
        u2             constant_pool_count;
        cp_info        constant_pool[constant_pool_count-1];
        u2             access_flags;
        u2             this_class;
        u2             super_class;
        u2             interfaces_count;
        u2             interfaces[interfaces_count];
        u2             fields_count;
        field_info     fields[fields_count];
        u2             methods_count;
        method_info    methods[methods_count];
        u2             attributes_count;
        attribute_info attributes[attributes_count];
    }
```

Её вы можете найти в спецификации JVM Chapter 4.1 The ClassFile Structure

Тут все просто - слева указана размерность в байтах, а справа описание.

Разбирать байт-код мы будем в hexadecimal, где каждая цифра занимает 4 бита, а следовательно, на два байта - 4 цифры и на четыре байта - 8 цифр.

magic

magic - это значение, которое идентифицирует формат нашего класса. Он равен `0xCAFEBABE`, который имеет свою историю создания.

minor_version, major_version

Это версии вашего `class` файла. Если мы назовем `major_version` M и `minor_version` m, то получаем версию нашего `class` файла как `M.m`

Сейчас я сразу буду приводить примеры из примера "Hello World", чтобы посмотреть как они используются:

```
    cafe babe -- magic
    0000 -- minor_version
    0034 -- major_version
```

Его же мы можем видеть в дизассемблированном коде, но уже в десятичной системе счисления:

```
    ...
    public class Main
      minor version: 0
      major version: 52
      flags: ACC_PUBLIC, ...
```

constant_pool_count

Здесь указывается количество переменных в пуле констант. При этом, если вы решили писать код на чистом байт-коде, то вам обязательно нужно следить за его значением, так как если вы укажете не то значение, то вся программа полетит к чертям (проверено!).

Также следует не забывать, что вы должны писать туда `количество_переменных_в_пуле + 1`

Итого, получаем:

```
    cafe babe   -- magic
    0000 0034   -- version
    001d        -- constant_pool_count
```

constant_pool[]

Каждый тип переменной в пуле констант имеет свою структуру:

```
cp_info {
 u1 tag;
 u1 info[];
}
```

Здесь все нужно делать последовательно. Сначала считываем `tag` , чтобы узнать тип переменной и по типу этой переменной смотрим какую структуру имеет последующее его значение `info[]`

Таблица с тэгами можно найти в спецификации Table 4.3 Constant pool tags

Собственно, вот табличка:

Constant Type	Value
`CONSTANT_Class`	7
`CONSTANT_Fieldref`	9
`CONSTANT_Methodref`	10
`CONSTANT_InterfaceMethodref`	11
`CONSTANT_String`	8
`CONSTANT_Integer`	3
`CONSTANT_Float`	4
`CONSTANT_Long`	5
`CONSTANT_Double`	6
`CONSTANT_NameAndType`	12
`CONSTANT_Utf8`	1
`CONSTANT_MethodHandle`	15
`CONSTANT_MethodType`	16
`CONSTANT_InvokeDynamic`	18

Как ранее уже говорилось, каждый тип константы имеет свою структуру.

Вот, например, структура `CONSTANT_Class`:

```
CONSTANT_Class_info {
    u1 tag;
    u2 name_index;
}
```

Структура поля и метода:

```
CONSTANT_Fieldref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}

CONSTANT_Methodref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

Рассмотрим часть нашего кода:

```
cafe babe
0000 0034
001d -- constant_pool_count
0a00 0600 0f09 0010 0011 0800 12 ...
```

Итак, смотрим на структуру константы и узнаем, что первый байт отведен под тип константы. Здесь мы видим `0a` (10) - а, следовательно, это `CONSTANT_Methodref`

Смотрим его структуру:

```
CONSTANT_Methodref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

После одного байта для тэга, нам нужно еще 4 байта для `class_index` и `name_and_type_index`

```
cafe babe
0000 0034
001d -- constant_pool_count

0a 0006 000f -- CONSTANT_Methodref
0900 1000 1108 0012 ...
```

Отлично, мы нашли одну из значений пула констант. Идем дальше. Смотрим, `09` - значит тип `CONSTANT_Fieldref`

Получаем:

```
cafe babe
0000 0034
001d -- constant_pool_count

0a 0006 000f -- CONSTANT_Methodref
09 0010 0011 -- CONSTANT_Fieldref
08 0012 ...
```

Вам может показаться, что большинство типов имеет одинаковую форму, но это не так. Например, структура следующего типа выглядит так, `CONSTANT_String`:

```
    CONSTANT_String_info {
        u1 tag;
        u2 string_index;
    }
```

Все эти структуры можно посмотреть в Chapter 4.4 The Constant Pool

Теперь разберем, что значат типы внутри самого `info`

Методы, которые попадают под паттерн `*_index` обычно содержат адрес из таблицы пула констант. Например, `class_index` на значение с типом `CONSTANT_Class_info`, а `string_index` на `CONSTANT_Utf8_info`

Это же мы можем видеть в дизассемблированном коде:

```
    #1 = Methodref          #6.#15         // java/lang/Object."<init>":()V
    #2 = Fieldref           #16.#17        //
java/lang/System.out:Ljava/io/PrintStream;
    #3 = String             #18
```

```
    0a 0006 000f -- CONSTANT_Methodref
    09 0010 0011 -- CONSTANT_Fieldref
    08 0012      -- CONSTANT_String
```

Также можно выделить представление чисел и строк.

Про представление чисел можно прочитать начиная с главы 4.4.4, а мы пока разберем лишь строки, так как числа не входят в программу Hello World

Собственно, вот так представляется строка:

```
    CONSTANT_Utf8_info {
        u1 tag;
        u2 length;
        u1 bytes[length];
    }
```

Например, наш Hello World:

```
    01      -- tag
    000b    -- length
    48 65 6c 6c 6f 20 57 6f 72 6c 64    -- bytes[length] // H e l l o   W o r
l d
```

И если разбирать все дальше, то получим:

```
    -- [Constant Pool]

    -- methodref
    0a 0006 000f

    -- fieldref
    09 0010 0011

    -- string
    08 0012

    -- methodref
    0a 0013 0014

    -- Class
    07 0015

    -- Class
    07 0016

    -- Utf8
    01 0006
    3c 69 6e 69 74 3e

    -- Utf8
    01 0003
    28 29 56

    -- Utf8
    01 0004
    43 6f 64 65

    -- Utf8
    01 000f
    4c 69 6e 65 4e 75 6d
    62 65 72 54 61 62 6c 65

    -- Utf8
    01 0004
    6d 61 69 6e

    -- Utf8
    01 0016
    28 5b 4c 6a 61 76 61 2f 6c 61 6e 67
    2f 53 74 72 69 6e 67 3b 29 56

    -- Utf8
    01 000a
    53 6f 75 72 63 65 46 69 6c 65

    -- Utf8
    01 0009
    4d 61 69 6e 2e 6a 61 76 61

    -- NameAndType
    0c 0007 0008

```

```
    -- Class
    07 0017

    -- NameAndType
    0c 0018 0019

    -- Utf8
    01 000b
    48 65 6c 6c 6f 20 57 6f 72 6c 64

    -- Class
    07 001a

    -- NameAndType
    0c 001b 001c

    -- Utf8
    01 0004
    4d 61 69 6e

    -- Utf8
    01 0010
    6a 61 76 61 2f 6c 61 6e 67 2f 4f 62 6a 65 63 74

    -- Utf8
    01 0010
    6a 61 76 61 2f 6c 61 6e 67 2f 53 79 73 74 65 6d

    -- Utf8
    01 0003
    6f 75 74

    -- Utf8
    01 0015
    4c 6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74
    72 65 61 6d 3b

    -- Utf8
    01 0013
    6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74 72
    65 61 6d

    -- Utf8
    01 0007
    70 72 69 6e 74 6c 6e

    -- Utf8
    01 0015
    28 4c 6a 61 76 61 2f 6c 61 6e 67 2f 53 74 72 69
    6e 67 3b 29 56

    -- [Constant Pool END]
```

Также, мы можем сравнить его с дизассемблированным кодом:

```
    Constant pool:
```

```
   #1 = Methodref          #6.#15         // java/lang/Object."<init>":
()V
   #2 = Fieldref           #16.#17        //
java/lang/System.out:Ljava/io/PrintStream;
   #3 = String             #18            // Hello World
   #4 = Methodref          #19.#20  // java/io/PrintStream.println:
(Ljava/lang/String;)V
   #5 = Class              #21            // Main
   #6 = Class              #22            // java/lang/Object
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               main
  #12 = Utf8               ([Ljava/lang/String;)V
  #13 = Utf8               SourceFile
  #14 = Utf8               Main.java
  #15 = NameAndType        #7:#8          // "<init>":()V
  #16 = Class              #23            // java/lang/System
  #17 = NameAndType        #24:#25        // out:Ljava/io/PrintStream;
  #18 = Utf8               Hello World
  #19 = Class              #26            // java/io/PrintStream
  #20 = NameAndType        #27:#28        // println:
(Ljava/lang/String;)V
  #21 = Utf8               Main
  #22 = Utf8               java/lang/Object
  #23 = Utf8               java/lang/System
  #24 = Utf8               out
  #25 = Utf8               Ljava/io/PrintStream;
  #26 = Utf8               java/io/PrintStream
  #27 = Utf8               println
  #28 = Utf8               (Ljava/lang/String;)V
```

Тем самым проверив, что все совпадает, ведь по сути `javap` просто обрабатывает этот байт-код и показывает нам его в форматированном виде.

Пул констант нужен для инструкций. Например:

```
    public Main();
      descriptor: ()V
      flags: ACC_PUBLIC
      Code:
        stack=1, locals=1, args_size=1
           0: aload_0
           1: invokespecial #1  // ссылается на адрес 1 в пуле констант
           4: return
```

Подробнее обо всех типах в пуле констант можно узнать в [Chapter 4.4 The Constant Pool]

Идем дальше по структуре *ClassFile*

access_flags

Это битовая маска для свойств модификаторов

Flag Name	Value	Interpretation
`ACC_PUBLIC`	0x0001	Declared `public`; may be accessed from outside its package.
`ACC_FINAL`	0x0010	Declared `final`; no subclasses allowed.
`ACC_SUPER`	0x0020	Treat superclass methods specially when invoked by the *invokespecial* instruction.
`ACC_INTERFACE`	0x0200	Is an interface, not a class.
`ACC_ABSTRACT`	0x0400	Declared `abstract`; must not be instantiated.
`ACC_SYNTHETIC`	0x1000	Declared synthetic; not present in the source code.
`ACC_ANNOTATION`	0x2000	Declared as an annotation type.
`ACC_ENUM`	0x4000	Declared as an `enum` type.

this_class

Должна содержать адрес на `this` класса. В нашем случае, она находится по адресу 5:

```
    Constant pool:
       #1 = Methodref          #6.#15         // java/lang/Object."<init>":
()V
       #2 = Fieldref           #16.#17        //
java/lang/System.out:Ljava/io/PrintStream;
       #3 = String             #18            // Hello World
       #4 = Methodref          #19.#20        // java/io/PrintStream.println:
(Ljava/lang/String;)V
       #5 = Class              #21            // Main
       #6 = Class              #22            // java/lang/Object
      ...
```

Следует заметить, что структуру этой переменной должна соответствовать `CONSTANT_Class_info`

super_class

Адрес предка класса. В нашем случае, значение по адресу `6`. Ну, и также обязательным является структура значения `CONSTANT_Class_info`. Все классы по умолчанию наследуются от `java.lang.Object`

Тут интересно заметить, что, если мы не указываем суперкласс, то этот наш `class` должен представлять собой объект `Object` (в случае, если super_class имеет значение 0)

> If the value of the super_class item is zero, then this class file must represent the class Object, the only class or interface without a direct superclass.

Далее, я бы хотел заметить, что имена этих классов заданы в структуре константы `CONSTANT_Utf8_info`. Если мы посмотрим ячейки `#21` и `#22`, то увидим:

```
    ...
    #21 = Utf8               Main
    #22 = Utf8               java/lang/Object
    ...
```

То есть в этих ячейках указан `name_index` из структуры:

```
    CONSTANT_Class_info {
        u1 tag;
        u2 name_index;
    }
```

interfaces_count, fields_count

Их в нашей программе нет, поэтому их значения будут равны 0000, а последующих значений **fields[], interfaces[]** просто не будет.

Читайте подробнее 4.1 The ClassFile Structure

methods_count

Количество методов. Хоть и в коде мы видим один метод в классе, но, на самом деле, их два. Кроме `main` метода еще есть конструктор по умолчанию. Поэтому их количество равно двум, в нашем случае.

methods[]

Каждый элемент должен соответствовать структуре *method_info* описанной в Chapter 4.6 Methods

```
    method_info {
        u2             access_flags;
        u2             name_index;
        u2             descriptor_index;
        u2             attributes_count;
        attribute_info attributes[attributes_count];
    }
```

В нашем байт-коде (отформатированном, с комментариями) выглядит это так:

```
    -- [methods]

    -- public Main();

    0001 --access_flags
    0007 -- name_index
    0008 -- descriptor_index
    0001 -- attributes_count

    -- attribute_info
```

```
    0009 -- attribute_name_index (Code)
    0000 001d - attribute_length
    0001 -- max_stack
    0001 -- max_locals
    0000 0005 -- code_length
    2a b7 00 01 b1  -- code[]

    0000 -- exception_table_length
    0001 --  attributes_count;
    000a -- attribute_name_index
    0000 0006 -- attribute_length
    00 01 00 00 00 01


    -- public static void main(java.lang.String...);

    0089  --access_flags
    000b  -- name_index
    000c  -- descriptor_index
    0001  -- attributes_count

    -- attribute_info
    0009  -- attribute_name_index (Code)
    0000 0025 -- attribute_length
    0002 -- max_stack
    0001 -- max_locals
    0000 0009 -- code_length
    b2 00 02 12 03 b6 00 04 b1 -- code[]

    0000 -- exception_table_length
    0001 -- attributes_count
    000a -- attribute_name_index
    0000 000a -- attribute_length
    00 02 00 00 00 04 00 08 00 05

    -- [methods END]
```

Разберем по-подробнее структуру методов:

access_flags

Маска модификаторов. Table 4.5 Method access and property flags

Flag Name	Value	Interpretation
`ACC_PUBLIC`	0x0001	Declared `public`; may be accessed from outside its package.
`ACC_PRIVATE`	0x0002	Declared `private`; accessible only within the defining class.
`ACC_PROTECTED`	0x0004	Declared `protected`; may be accessed within subclasses.
`ACC_STATIC`	0x0008	Declared `static`.
`ACC_FINAL`	0x0010	Declared `final`; must not be overridden (§5.4.5).
`ACC_SYNCHRONIZED`	0x0020	Declared `synchronized`; invocation is wrapped by a monitor use.
`ACC_BRIDGE`	0x0040	A bridge method, generated by the compiler.
`ACC_VARARGS`	0x0080	Declared with variable number of arguments.
`ACC_NATIVE`	0x0100	Declared `native`; implemented in a language other than Java.
`ACC_ABSTRACT`	0x0400	Declared `abstract`; no implementation is provided.
`ACC_STRICT`	0x0800	Declared `strictfp`; floating-point mode is FP-strict.
`ACC_SYNTHETIC`	0x1000	Declared synthetic; not present in the source code.

Как мы можем видеть из байт-кода, в методе `public Main();` (конструктор) стоит маска `0001`, который означает `ACC_PUBLIC`.

А теперь сами попробуем собрать метод `main` . Вот что у него есть:

- public - ACC_PUBLIC - 0x0001
- static - ACC_STATIC - 0x0008
- String ... args - ACC_VARARGS - 0x0080

Собираем маску: 0x0001 + 0x0008 + 0x0080 = **0x0089** . Итак, мы получили `access_flag`

> К слову, ACC_VARARGS здесь необязательный, в том плане, что, если бы мы использовали String[] args вместо String ... args, то этого флага бы не было

name_index

Адрес имени метода (`CONSTANT_Utf8_info`) в пуле констант. Здесь важно заметить, что имя конструктора это не Main, а `<init>`, расположенная в ячейке #7.

Подробнее о `<init>` и `<clinit>` в Chapter 2.9 Special Methods

descriptor_index

Грубо говоря, это адрес указывающий на дескриптор метода. Этот дескриптор содержит тип возвращаемого значения и тип его сигнатуры.

Также, в JVM используются интерпретируемые сокращения:

BaseType **Character**	**Type**	**Interpretation**
`B`	`byte`	signed byte
`C`	`char`	Unicode character code point in the Basic Multilingual Plane, encoded with UTF-16
`D`	`double`	double-precision floating-point value
`F`	`float`	single-precision floating-point value
`I`	`int`	integer
`J`	`long`	long integer
`L` *ClassName* `;`	`reference`	an instance of class *ClassName*
`S`	`short`	signed short
`Z`	`boolean`	`true` or `false`
`[`	`reference`	one array dimension

В общем случае это выглядит так:

```
    ( ParameterDescriptor* ) ReturnDescriptor
```

Например, следующий метод:

```
    Object m(int i, double d, Thread t) {..}
```

Можно представить в виде

```
    (IDLjava/lang/Thread;)Ljava/lang/Object
```

Собственно, `I` - это `int` , `D` - это `double`, а `Ljava/lang/Thread;` класс `Thread` из стандартной библиотеки.

Далее, идут атрибуты, которые также имеют свою структуру.

Но сначала, как и всегда, идет его количество `attributes_count`

Затем сами атрибуты со структурой описанной в Chapter 4.7 Attributes

```
    attribute_info {
        u2 attribute_name_index;
        u4 attribute_length;
        u1 info[attribute_length];
    }
```

attribute_name_index

Указание имени атрибута. В нашем случае, у обоих методов это `Code`. Атрибуты это отдельная большая тема, в котором можно по спецификации создавать даже свои атрибуты. Но нам пока следует знать, что `attribute_name_index` просто указывает на адрес в пуле констант со структурой `CONSTANT_Utf8_info`

attribute_length

Содержит длину атрибута, не включая `attribute_name_index` и `attribute_length`

info

Далее, мы будем использовать структуру `Code`, так как в значении `attribute_name_index` мы указали на значение в пуле констант `Code`.

Подробнее: Chapter 4.7.3 The Code Attribute

Вот его структура:

```
    Code_attribute {
        u2 attribute_name_index;
        u4 attribute_length;
        u2 max_stack;
        u2 max_locals;
        u4 code_length;
        u1 code[code_length];
        u2 exception_table_length;
        {   u2 start_pc;
            u2 end_pc;
            u2 handler_pc;
            u2 catch_type;
        } exception_table[exception_table_length];
        u2 attributes_count;
        attribute_info attributes[attributes_count];
    }
```

max_stack

Максимальный размер стека нужный для операции.

На тему стека можно почитать "О стеке и куче в контексте мира Java" или в "JVM Internals"

max_locals

Максимальный размер локальных переменных

Ознакомится с локальными переменными можно либо в Mastering Java Bytecode at the Core of the JVM или в том же JVM Internals

code_length

Размер кода, который будет исполнятся внутри метода

code[]

Каждый код указывает на какую-то инструкцию. Таблицу соотношения **optcode** и команды с мнемоникой можно найти в википедии - Java bytecode instruction listings

Для примера, возьмем наш конструктор:

```
    -- public Main();

    0001 --access_flags
    0007 -- name_index
    0008 -- descriptor_index
    0001 -- attributes_count

    -- attribute_info
    0009 -- attribute_name_index (Code)
    0000 001d - attribute_length
    00 01 -- max_stack
    00 01 -- max_locals
    00 00 00 05 -- code_length
    2a b7 00 01 b1  -- code[]

    0000 -- exception_table_length
    0001 --  attributes_count;
    00 0a -- attribute_name_index
    0000 0006 -- attribute_length
    00 01 00 00 00 01
```

Здесь мы можем найти наш код:

```
    2a b7 00 01 b1
```

Ищем в таблице команды и сопоставляем:

```
    2a - aload_0
    b7 0001 - invokespecial #1
    b1 - return
```

Также описания этих команд можно найти здесь: Chapter 4.10.1.9. Type Checking Instructions

exception_table_length

Задает число элементов в таблице exception_table. У нас пока нет перехватов исключений поэтому разбирать его не будем. Но дополнительно можно почитать Chapter 4.7.3 The Code Attribute

exception_table[]

Имеет вот такую структуру:

```
    {
        u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
    }
```

Если упрощать, то нужно указать начало, конец (`start_pc`, `end_pc`) кода, который будет обрабатывать `handler_pc` и тип исключения `catch_type`

attributes_count

Количество атрибутов в `Code`

attributes[]

Атрибуты, часто используются анализаторами или отладчиками.

Конец

Вот мы и разобрали простую программку Hello World:

Листинг байт-кода с комментариями можно найти на моем гисте: gist.github

Использованная литература

- The Java® Virtual Machine Specification - docs.oracle