1. Платформа: история и окружение

1.1. В этом разделе

Краткая история (причины возникновения); инструментарий, выбор версии; CLI; структура проекта; документирование; некоторые интересные способы сборки проектов.

В этом разделе происходит первое знакомство со внутреннем устройством языка Java и фреймворком разработки приложений с его использованием. Рассматривается примитивный инструментарий и базовые возможности платформы для разработки приложений на языке Java. Разбирается структура проекта, а также происходит ознакомление с базовым инструментарием для разработки на Java.

— JDK
— JRE
— JVM
— JIT
— CLI
— Docker

1.2. Краткая история (причины возникновения)

— Язык создавали для разработки встраиваемых систем, сетевых приложений и прикладного ПО;
— Популярен из-за скорости исполнения и полного абстрагирования от исполнителя кода;
— Часто используется для программирования бэк-энда веб-приложений из-за изначальной нацеленности на сетевые приложения.

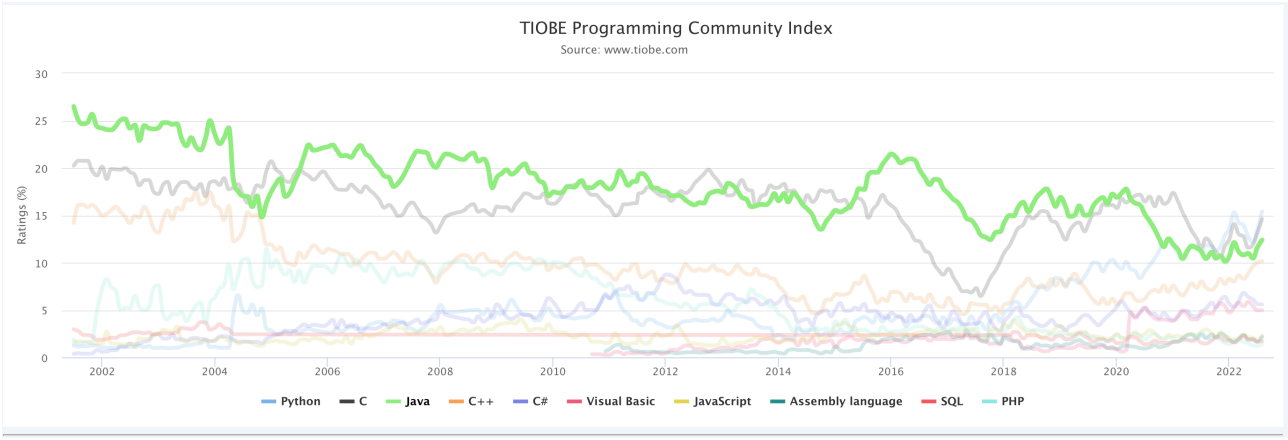


Рис. 1: График популярности языков программирования TIOBE

1.2.1. Задания для самопроверки

1. Как Вы думаете, почему язык программирования Java стал популярен в такие короткие сроки?
   — существовавшие на тот момент Pascal и C++ были слишком сложными;

— Java быстрее C++;
— Однажды написанная на Java программа работает везде.

1.3. Базовый инструментарий, который понадобится (выбор IDE)

— NetBeans - хороший, добротный инструмент с лёгким ностальгическим оттенком;
— Eclipse - для поклонников Eclipse Foundation и швейцарских ножей с полусотней лезвий;
— IntelliJ IDEA - стандарт де-факто, используется на курсе и в большинстве современных компаний;
— Android Studio - если заниматься мобильной разработкой.

1.3.1. Задания для самопроверки

1. Как Вы думаете, почему среда разработки IntelliJ IDEA стала стандартом де-факто в коммерческой разработке приложений на Java?
   — NetBeans перестали поддерживать;
   — Eclipse слишком медленный и тяжеловесный;
   — IDEA оказалась самой дружелюбной к начинающему программисту;
   — Все варианты верны.

1.4. Что нужно скачать, откуда (как выбрать вендора, версии)

Для разработки понадобится среда разработки (IDE) и инструментарий разработчика (JDK). JDK выпускается несколькими поставщиками, большинство из них бесплатны и полнофункциональны, то есть поддерживают весь функционал языка и платформы.

В последнее время, с развитием контейнеризации приложений, часто устанавливают инструментарий в Docker-контейнер и ведут разработку прямо в контейнере, это позволяет не захламлять компьютер разработчика разными версиями инструментария и быстро разворачивать свои приложения в CI или на целевом сервере.

В общем случае, для разработки на любом языке программирования нужны так называемые SDK (Software Development Kit, англ. - инструментарий разработчика приложений или инструментарий для разработки приложений). Частный случай такого SDK - инструментарий разработчика на языке Java - Java Development Kit.

На курсе будет использоваться BellSoft Liberica JDK 11, но возможно использовать и других производителей, например, самую распространённую Oracle JDK. Производителя следует выбирать из требований по лицензированию, так, например, Oracle JDK можно использовать бесплатно только в личных целях, за коммерческую разработку с использованием этого инструментария придётся заплатить.

Для корректной работы самого инструментария и сторонних приложений, использующих инструментарий, проследите, пожалуйста, что установлены следующие переменные среды ОС:

- в системную `PATH` добавить путь до исполняемых файлов JDK, например, для UNIX-подобных систем: `PATH=$PATH:/usr/lib/jvm/jdk1.8.0_221/bin`
- `JAVA_HOME` путь до корня JDK, например, для UNIX-подобных систем: `JAVA_HOME=/usr/lib/jvm/jdk1.8.0_221/`
- `JRE_HOME` путь до файлов JRE из состава установленной JDK, например, для UNIX-подобных систем: `JRE_HOME=/usr/lib/jvm/jdk1.8.0_221/jre/`
- `J2SDKDIR` устаревшая переменная для JDK, используется некоторыми старыми приложениями, например, для UNIX-подобных систем: `J2SDKDIR=/usr/lib/jvm/jdk1.8.0_221/`
- `J2REDIR` устаревшая переменная для JRE, используется некоторыми старыми приложениями, например, для UNIX-подобных систем: `J2REDIR=/usr/lib/jvm/jdk1.8.0_221/jre/`

Также возможно использовать и другие версии, но не старше 1.8. Это обосновано тем, что основные разработки на данный момент только начинают обновлять инструментарий до более новых версий (часто 11 или 13) или вовсе переходят на другие JVM-языки, такие как Scala, Groovy или Kotlin.

Иногда для решения вопроса менеджмента версий прибегают к стороннему инструментарию, такому как SDKMan.

Для решения некоторых несложных задач курса мы будем писать простые приложения, не содержащие ООП, сложных взаимосвязей и проверок, в этом случае нам понадобится Jupyter notebook с установленным ядром (kernel) IJava.

1.4.1. Задания для самопроверки

1. Чем отличается SDK от JDK?
2. Какая версия языка (к сожалению) остаётся самой популярной в разработке на Java?
3. Какие ещё JVM языки существуют?

1.5. Из чего всё состоит (JDK, JRE, JVM и их друзья)

TL;DR:

- JDK = JRE + инструменты разработчика;
- JRE = JVM + библиотеки классов;
- JVM = Native API + механизм исполнения + управление памятью.

Как именно всё работает? Если коротко, то слой за слоем накладывая абстракции. Программы на любом языке программирования исполняются на компьютере, то есть, так или иначе, задействуют процессор, оперативную память и прочие аппаратурные компоненты. Эти аппаратурные компоненты предоставляют для доступа к себе низкоуровневые интерфейсы, которые задействует операционная система, предоставляя в свою очередь ин-

терфейс чуть проще программам, взаимодействующим с ней. Этот интерфейс взаимодействия с ОС мы для простоты будем называть Native API.

С ОС взаимодействует JVM (Wikipedia: Список виртуальных машин Java), то есть, используя Native API, нам становится всё равно, какая именно ОС установлена на компьютере, главное уметь выполняться на JVM. Это открывает простор для создания целой группы языков, они носят общее бытовое название JVM-языки, к ним относят Scala, Groovy, Kotlin и другие. Внутри JVM осуществляется управление памятью, существует механизм исполнения программ, специальный JIT[1]-компилятор, генерирующий платформенно-зависимый код.

JVM для своей работы запрашивает у ОС некоторый сегмент оперативной памяти, в котором хранит данные программы. Это хранение происходит «слоями»:

1. Eden Space (heap) – в этой области выделяется память под все создаваемые из программы объекты. Большая часть объектов живёт недолго (итераторы, временные объекты, используемые внутри методов и т.п.), и удаляются при выполнении сборок мусора это области памяти, не перемещаются в другие области памяти. Когда данная область заполняется (т.е. количество выделенной памяти в этой области превышает некоторый заданный процент), сборщик мусора выполняет быструю (minor collection) сборку. По сравнению с полной сборкой, она занимает мало времени, и затрагивает только эту область памяти, а именно, очищает от устаревших объектов Eden Space и перемещает выжившие объекты в следующую область.
2. Survivor Space (heap) – сюда перемещаются объекты из предыдущей области после того, как они пережили хотя бы одну сборку мусора. Время от времени долгоживущие объекты из этой области перемещаются в Tenured Space.
3. Tenured (Old) Generation (heap) — Здесь скапливаются долгоживущие объекты (крупные высокоуровневые объекты, синглтоны, менеджеры ресурсов и прочие). Когда заполняется эта область, выполняется полная сборка мусора (full, major collection), которая обрабатывает все созданные JVM объекты.
4. Permanent Generation (non-heap) – Здесь хранится метаинформация, используемая JVM (используемые классы, методы и т.п.).
5. Code Cache (non-heap) — эта область используется JVM, когда включена JIT-компиляция, в ней кешируется скомпилированный платформенно-зависимый код.

JVM самостоятельно осуществляет сборку так называемого мусора, что значительно облегчает работу программиста по отслеживанию утечек памяти, но важно помнить, что в Java утечки памяти всё равно существуют, особенно при программировании многопоточных приложений.

[1] JIT, just-in-time - англ. во́время, прямо сейчас

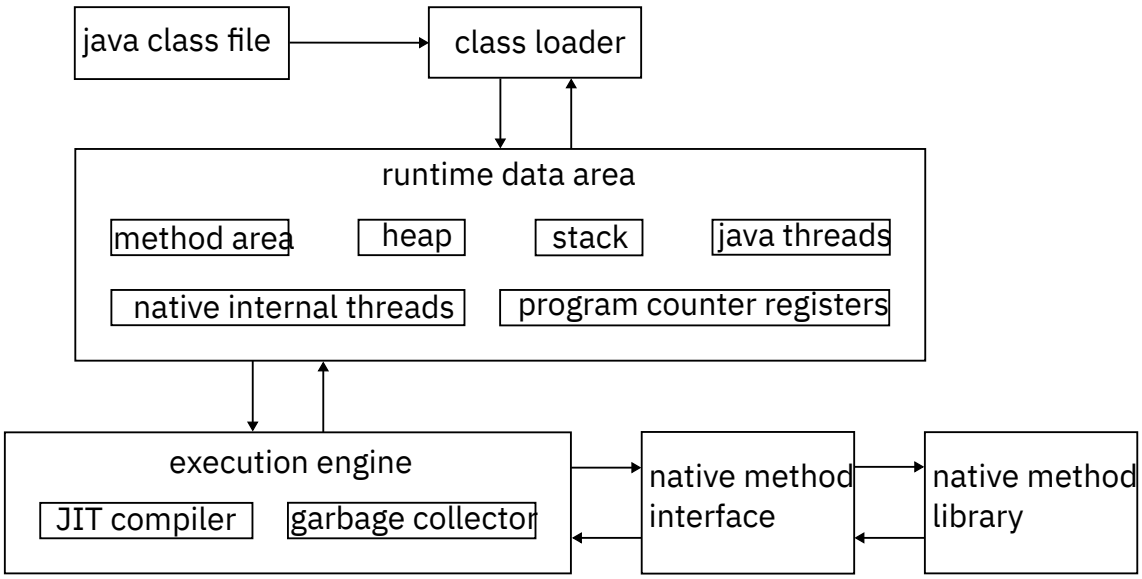


Рис. 2: принцип работы JVM

На пользовательском уровне важно не только исполнять базовые инструкции программы, но чтобы эти базовые инструкции умели как-то взаимодействовать со внешним миром, в том числе другими программами, поэтому JVM интегрирована в JRE - Java Runtime Environment. JRE - это набор из классов и интерфейсов, реализующих

- возможности сетевого взаимодействия;
- рисование графики и графический пользовательский интерфейс;
- мультимедиа;
- математический аппарат;
- наследование и полиморфизм;
- рефлексию;
- ... многое другое.

Java Development Kit является изрядно дополненным специальными Java приложениями SDK. JDK дополняет JRE не только утилитами для компиляции, но и утилитами для создания документации, отладки, развёртывания приложений и многими другими. В таблице 1.5 на странице 6, приведена примерная структура и состав JDK и JRE, а также указаны их основные и наиболее часто используемые компоненты из состава Java Standard Edition. Помимо стандартной редакции существует и Enterprise Edition, содержащий компоненты для создания веб-приложений, но JEE активно вытесняется фреймворками Spring и Spring Boot.

Java Development Kit									
Java Runtime Environment									
Java Standard Edition									
Language									
tools + tools api	javac	java	javadoc	javap	jar	JPDA			
	JConsole	JavaVisualVM	JMC	JFR	Java DB	Int'l	JVM TI		
	IDL	Troubleshoot	Security	RMI	Scripting	Web services	Deploy		
deployment	Java Web				Applet/Java plug-in				
UI toolkit	Swing		Java 2D		AWT	Accessibility			
	Drag'n'Drop		Input Methods		Image I/O	Print Service	Sound		
Integration libraries	IDL	JDBC	JNDI	RMI	RMI-IIOP	Scripting			
Other base libraries	Override Mechanism		Intl Support		Input/Output		JMX		
	XML JAXP		Math		Networking		Beans		
	Security		Serialization		Extension Mechanism		JNI		
Java lang and util base libs	JAR	Lang and util	Ref Objects		Preference API		Reflection		
	Zip	Management	Instrumentation		Stream API		Collections		
	Logging	Regular Expressions	Concurrency Utilities		Datetime		Versioning		
JVM	Java Hot Spot VM (JIT)								

Таблица 1: Общее представление состава JDK

1.5.1. Задания для самопроверки

1. JVM и JRE - это одно и тоже?
2. Что входит в состав JDK, но не входят в состав JRE?

3. Утечки памяти
    - Невозможны, поскольку работает сборщик мусора;
    - Возможны;
    - Существуют только в C++ и других языках с открытым менеджментом памяти.

1.6. Структура проекта (пакеты, классы, метод main, комментарии)

Проекты могут быть любой сложности. Часто структуру проекта задаёт сборщик проекта, предписывая в каких папках будут храниться исходные коды, исполняемые файлы, ресурсы и документация. Без их использования необходимо задать структуру самостоятельно.

Простейший проект чаще всего состоит из одного файла исходного кода, который возможно скомпилировать и запустить как самостоятельный объект. Отличительная особенность в том, что чаще всего это один или несколько статических методов в одном классе.

Файл `Main.java` в этом случае может иметь следующий, минималистичный вид

```java
public class Main {
   public static void main(String[] args) {
      System.out.println("Hello, world!");
   }
}
```

Скриптовый проект это достаточно новый тип проектов, он получил развитие благодаря растущей популярности Jupyter Notebook. Скриптовые проекты удобны, когда нужно отработать какую-то небольшую функциональность или пошагово пояснить работу какого-то алгоритма.

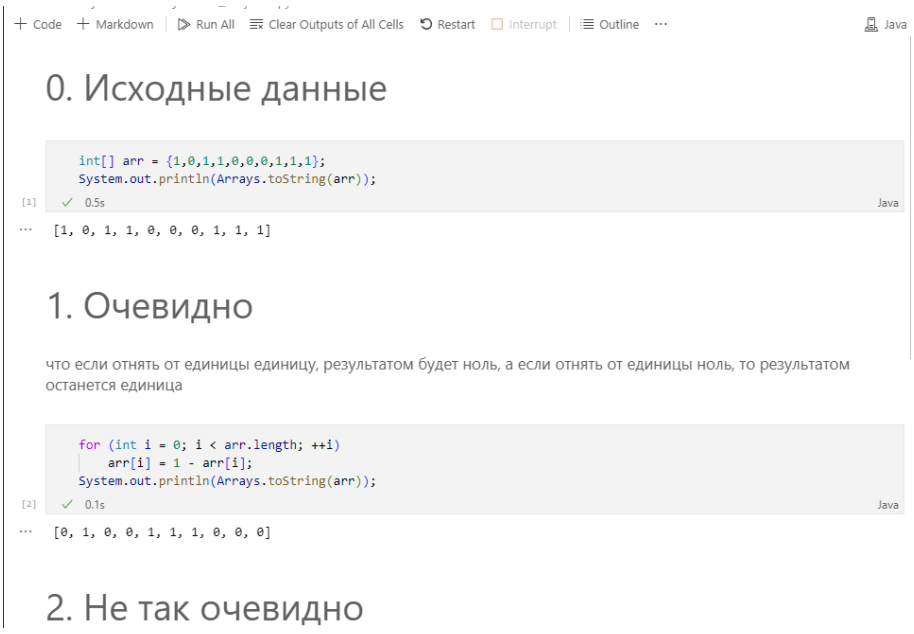


Рис. 3: Пример простого Java проекта в Jupyter Notebook

Обычный проект состоит из пакетов, которые содержат классы, которые в свою очередь как-то связаны между собой и содержат код, который исполняется.

— Пакеты. Пакеты объединяют классы по смыслу. Классы, находящиеся в одном пакете доступны друг другу даже если находятся в разных проектах. У пакетов есть правила именования: обычно это обратное доменное имя (например, для gb.ru это будет ru.gb), название проекта, и далее уже внутренняя структура. Пакеты именуют строчными латинскими буквами. Чтобы явно отнести класс к пакету, нужно прописать в классе название пакета после оператора `package`.
— Классы. Основная единица исходного кода программы. Одному файлу следует сопоставлять один класс. Название класса - это имя существительное в именительном падеже, написанное с заглавной буквы. Если требуется назвать класс в несколько слов, применяют UpperCamelCase.
— `public static void main(String[] args)`. Метод, который является точкой входа в программу. Должен находиться в публичном классе. При создании этого метода важно полностью повторить его сигнатуру и обязательно написать его с название со строчной буквы.
— Комментарии. Это часть кода, которую игнорирует компилятор при преобразовании исходного кода. Комментарии бывают:
  – `// comment` - до конца строки. Самый простой и самый часто используемый комментарий.
  – `/* comment */` - внутристрочный или многострочный. Никогда не используйте его внутри строк, несмотря на то, что это возможно.
  – `/** comment */` - комментарий-документация. Многострочный. Из него утилитой Javadoc создаётся веб-страница с комментарием.

Для примера был создан проект, содержащий два класса, находящихся в разных пакетах. Дерево проекта представлено на рис. 1.6, где папка с выходными (скомпилированными) бинарными файлами пуста, а файл README.md создан для лучшей демонстрации корня проекта.

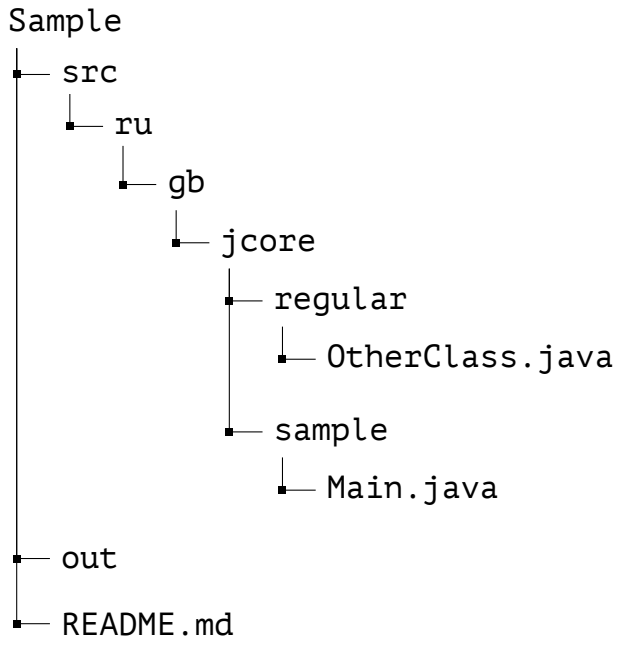


Рис. 4: Структура простого проекта

Содержимое файлов исходного кода представлено ниже.

```
package ru.gb.jcore.sample;

import ru.gb.jcore.regular.OtherClass;

public class Main {
   public static void main(String[] args) {
      System.out.println("Hello, world!"); // greetings
      int result = OtherClass.sum(2, 2); // using a class from other package
      System.out.println(OtherClass.decorate(result));
   }
}
```

```
package ru.gb.jcore.regular;

public class OtherClass {
   public static int sum(int a, int b) {
      return a + b; // return without overflow check
   }

   public static String decorate(int a) {
      return String.format("Here is your number: %d.", a);
   }
}
```

1.6.1. Задания для самопроверки

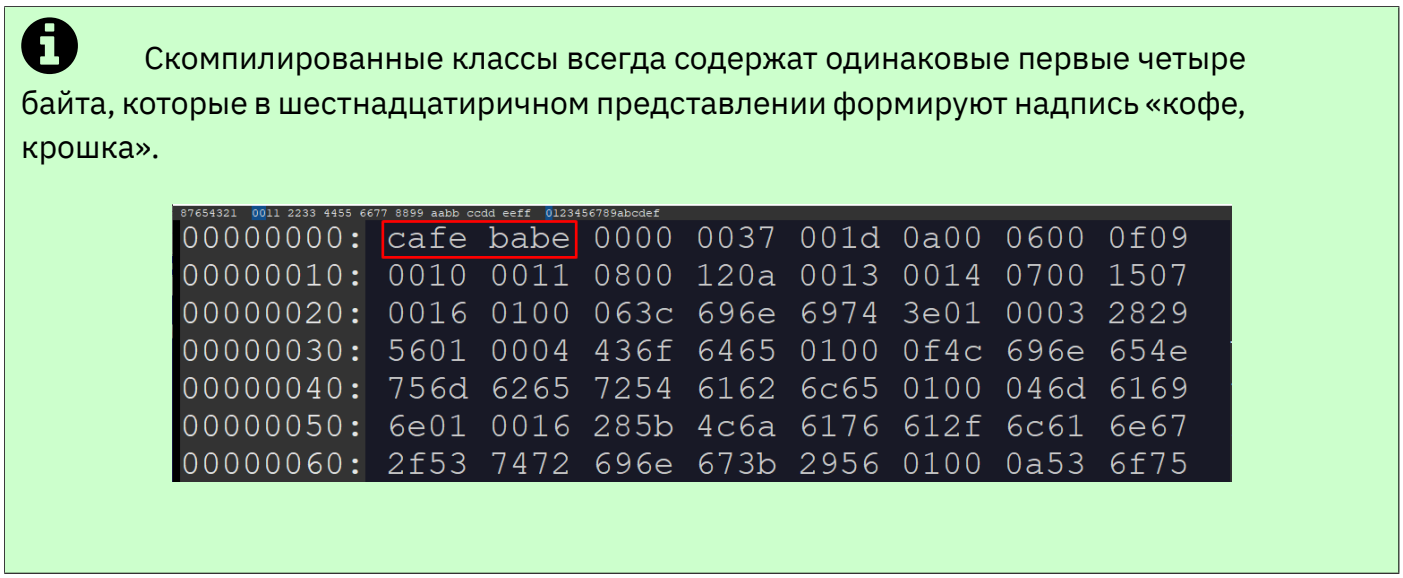
1. Зачем складывать классы в пакеты?
2. Может ли существовать класс вне пакета?
3. Комментирование кода
   - Нужно только если пишется большая подключаемая библиотека;
   - Хорошая привычка;
   - Захламляет исходники.

1.7. Отложим мышки в сторону (CLI: сборка, пакеты, запуск)

Простейший проект возможно скомпилировать и запустить без использования тяжеловесных сред разработки, введя в командной строке ОС две команды:

- javac <Name.java> скомпилирует файл исходников и создаст в этой же папке файл с байт-кодом;
- java Name запустит скомпилированный класс (из файла с расширением `.class`).

```
ivan-igorevich@gb sources % ls
Main.java
ivan-igorevich@gb sources % javac Main.java
ivan-igorevich@gb sources % ls
Main.class  Main.java
ivan-igorevich@gb sources % java Main
Hello, world!
```



Скомпилированные классы всегда содержат одинаковые первые четыре байта, которые в шестнадцатиричном представлении формируют надпись «кофе, крошка».

```
87654321  0011 2233 4455 6677 8899 aabb ccdd eeff  0123456789abcdef
00000000: cafe babe 0000 0037 001d 0a00 0600 0f09
00000010: 0010 0011 0800 120a 0013 0014 0700 1507
00000020: 0016 0100 063c 696e 6974 3e01 0003 2829
00000030: 5601 0004 436f 6465 0100 0f4c 696e 654e
00000040: 756d 6265 7254 6162 6c65 0100 046d 6169
00000050: 6e01 0016 285b 4c6a 6176 612f 6c61 6e67
00000060: 2f53 7472 696e 673b 2956 0100 0a53 6f75
```

Для компиляции более сложных проектов, необходимо указать компилятору, откуда забирать файлы исходников и куда складывать готовые файлы классов, а интерпретатору, откуда забирать файлы скомпилированных классов. Для этого существуют следующие ключи:

- javac:
  - `-d` выходная папка (директория) назначения;
  - `-sourcepath` папка с исходниками проекта;
- java:
  - `-classpath` папка с классами проекта;

Классы проекта компилируются в выходную папку с сохранением иерархии пакетов.

```
ivan-igorevich@gb Sample % javac -sourcepath ./src -d out src/ru/gb/jcore/sample/Main.java
ivan-igorevich@gb Sample % java -classpath ./out ru.gb.jcore.sample.Main
Hello, world!
Here is your number: 4.
```

1.7.1. Задания для самопроверки

1. Что такое javac?
2. Кофе, крошка?
3. Где находится класс в папке назначения работы компилятора?
   - В подпапках, повторяющих структуру пакетов в исходниках
   - В корне плоским списком;
   - Зависит от ключей компиляции.

1.8. Документирование (Javadoc)

Документирование конкретных методов и классов всегда ложится на плечи программиста, потому что никто не знает программу и алгоритмы в ней лучше, чем программист. Утилита Javadoc избавляет программиста от необходимости осваивать инструменты создания веб-страниц и записывать туда свою документацию. Достаточно писать хорошо отформатированные комментарии, а остальное Javadoc возьмёт на себя.

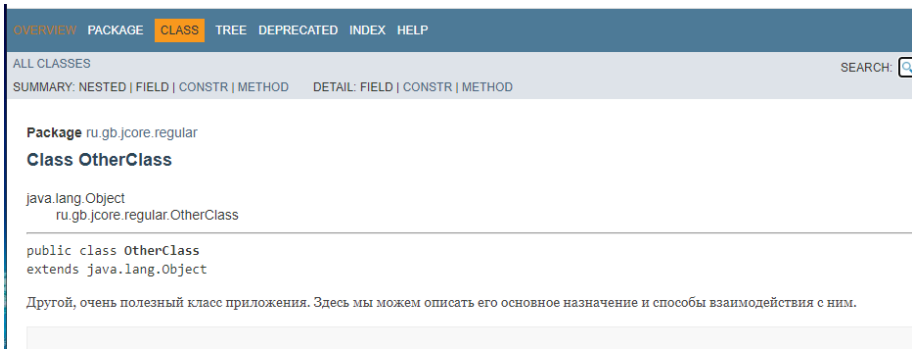


Рис. 5: Часть страницы автосгенерированной документации

Чтобы просто создать документацию надо вызвать утилиту javadoc с набором ключей.

— `ru` пакет, для которого нужно создать документацию;
— `-d` папка (или директория) назначения;
— `-sourcepath` папка с исходниками проекта;
— `-cp` путь до скомпилированных классов;
— `-subpackages` нужно ли заглядывать в пакеты-с-пакетами;

Часто необходимо указать, в какой кодировке записан файл исходных кодов, и в какой кодировке должна быть выполнена документация (например, файлы исходников на языке Java всегда сохраняются в кодировке UTF-8, а основная кодировка для ОС Windows - cp1251)

— `-locale ru_RU` язык документации (для правильной расстановки переносов и разделяющих знаков);
— `-encoding` кодировка исходных текстов программы;
— `-docencoding` кодировка конечной сгенерированной документации.

Чаще всего в комментариях используются следующие ключевые слова:

— `@param` описание входящих параметров
— `@throws` выбрасываемые исключения
— `@return` описание возвращаемого значения
— `@see` где ещё можно почитать по теме
— `@since` с какой версии продукта доступен метод
— `{@code "public"}` вставка кода в описание

1.8.1. Задания для самопроверки

1. Javadoc находится в JDK или JRE?
2. Что делает утилита Javadoc?
   — Создаёт комментарии в коде;
   — Создаёт программную документацию;
   — Создаёт веб-страницу с документацией из комментариев.

1.9. Автоматизируй это (Makefile, Docker)

В подразделе 1.7 мы проговорили о сборке проектов вручную. Компилировать проект таким образом — занятие весьма утомительное, особенно когда исходных файлов стано-

вится много, в проект включаются библиотеки и прочее.

`Makefile` — это набор инструкций для программы `make` (классическая, это GNU Automake), которая помогает собирать программный проект в одну команду. Если запустить `make` то программа попытается найти файл с именем по-умолчанию `Makefile` в текущем каталоге и выполнить инструкции из него.

Make, не привносит ничего принципиально нового в процесс компиляции, а только лишь автоматизируют его. В простейшем случае, в `Makefile` достаточно описать так называемую цель, `target`, и что нужно сделать для достижения этой цели. Цель, собираемая по-умолчанию называется `all`, так, для простейшей компиляции нам нужно написать:

```
1 all:
2     javac -sourcepath .src/ -d out src/ru/gb/jcore/sample/Main.java
```

Внимание поклонникам войны за пробелы против табов в тексте программы: в Makefile для отступов при описании таргетов нельзя использовать пробелы. Только табы. Иначе make обнаруживает ошибку синтаксиса.

По сути, это всё. Но возможно сделать более гибко настраиваемый файл, чтобы не нужно было запоминать, как называются те или иные папки и файлы. В Makefile можно записывать переменные, например:

— SRCDIR := src
— OUTDIR := out

И далее вызывать их (то есть подставлять их значения в нужное место текста) следующим образом:

```
1 javac -sourcepath .${SRCDIR}/ -d ${OUTDIR}
```

Чтобы вызвать утилиту для сборки цели по-умолчанию, достаточно в папке, содержащей `Makefile` в терминале написать `make`. Чтобы воспользоваться другими написанными таргетами нужно после имени утилиты написать через пробел название таргета

Docker — программное обеспечение для автоматизации развёртывания и управления приложениями, контейнеризатор приложений. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть развёрнут на любой системе, поддерживающей соответствующую технологию.

Docker также не привносит ничего технологически нового, но даёт возможность не устанавливать JDK и не думать о переключении между версиями, достаточно взять контейнер с нужной версией инструментария и запустить приложение в нём.

Образы и контейнеры создаются с помощью специального файла, имеющего название `Dockerfile`. Первой строкой `Dockerfile` мы обязательно должны указать, какой виртуальный образ будет для нас основой. Здесь можно использовать как образы ОС, так и образы SDK.

```
FROM bellsoft/liberica-openjdk-alpine:11.0.16.1-1
```

При создании образа необходимо скопировать все файлы из папки `src` проекта внутрь образа, в папку `src`.

```
COPY ./src ./src
```

Потом, также при создании образа, надо будет создать внутри папку `out` простой терминальной командой, чтобы компилятору было куда складывать готовые классы.

```
RUN mkdir ./out
```

Последнее, что будет сделано при создании образа - запущена компиляция.

```
RUN javac -sourcepath ./src -d out ./src/ru/gb/dj/Main.java
```

Последняя команда в `Dockerfile` говорит, что нужно сделать, когда контейнер создаётся из образа и заускается.

```
CMD java -classpath ./out ru.gb.dj.Main
```

Docker-образ и, как следствие, Docker-контейнеры возможно настроить таким образом, чтобы скомпилированные файлы находились не в контейнере, а складывались обратно на компьютер пользователя через общие папки.

Часто команды разработчиков эмулируют таким образом реальный продакшн сервер, используя в качестве исходного образа не JDK, а образ целевой ОС, вручную устанавливают на ней JDK, запуская далее своё приложение.

Домашнее задание

— Создать проект из трёх классов (основной с точкой входа и два класса в другом пакете), которые вместе должны составлять одну программу, позволяющую производить четыре основных математических действия и осуществлять форматированный вывод результатов пользователю;
— Скомпилировать проект, а также создать для этого проекта стандартную веб-страницу с документацией ко всем пакетам;
— Создать Makefile с задачами сборки, очистки и создания документации на весь проект.
— *Создать два Docker-образа. Один должен компилировать Java-проект обратно в папку на компьютере подьзователя, а второй забирать скомпилированные классы и исполнять их.