Платформа: история и окружение

Иван Игоревич Овчинников

GeekBrains. Java Core.

2022

Добро пожаловать на техническую специализацию Java, лучшую из всех технических специализаций Java, что вы сможете найти.

Введение и знакомство (о себе)

Иван Овчинников. НПО ИТ, РКС, GB.

Используемые технологии

Много и с удовольствием C, C++, Java, Verilog. C# и Python даже не считаем.

Для GB более 20 потоков по разным направлениям, более двух тысяч студентов.

Здравствуйте, меня зовут Иван, я разработчик программного обеспечения в Российских космических системах, начальник группы программистов, автор и преподаватель нескольких курсов в направлении программирования на портале GeekBrains. Пара слов обо мне, почему я нахожусь здесь и планирую Вам что-то рассказывать про язык Java? Для этого придётся вынести за скобки мой опыт работы в цифровой схемотехнике и других языках, в том числе создание бортовой аппаратуры, которая прямо сейчас летает у нас над головами или продолжает доставлять другие полезные грузы в космос. С точки зрения языка Java, обо мне можно сказать, что я являюсь разработчиком нескольких отраслевых информационных систем поддержки единой наземной инфраструктуры российского космического агентства. На моём счету также участие в межотраслевых проектах по созданию единых баз данных космического применения.

О курсе

и об уркое

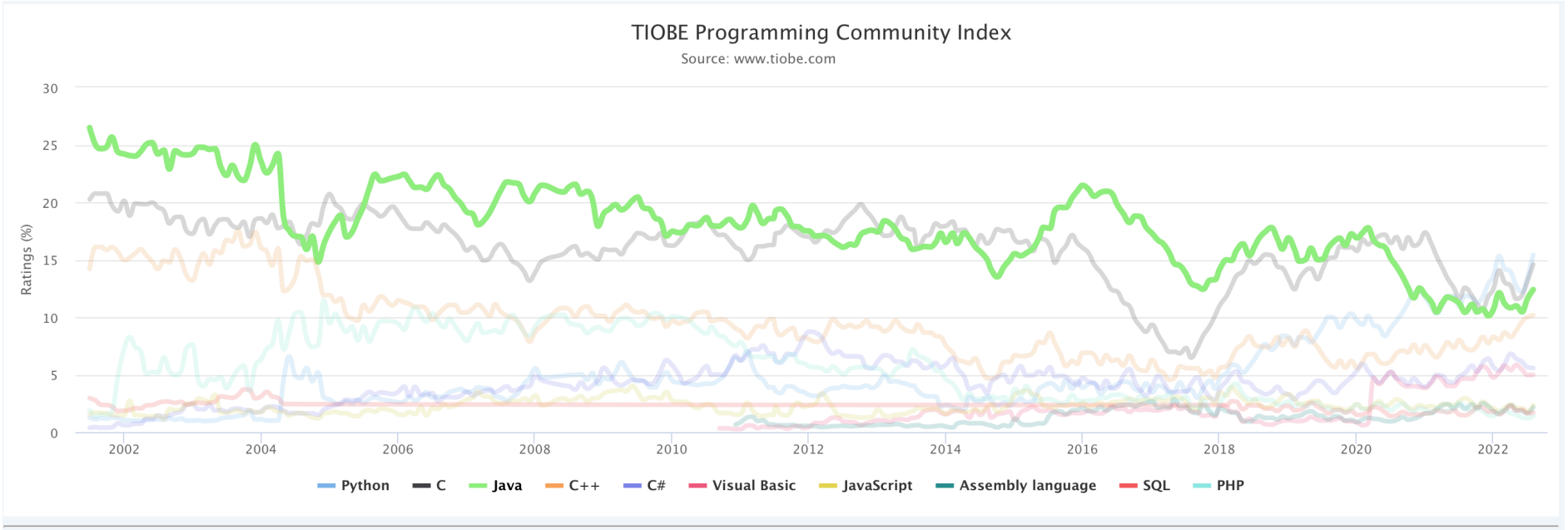
Данный курс, первый из технической специализации, направлен на первое знакомство со внутренним устройством языка и фреймворком разработки приложений с его использованием. Курс демонстрирует механизмы работы знакомых студенту концепций на примере языка Java. В рамках курса будут рассмотрено устройство языка Java и сопутствующих технологических решений, платформы для создания и запуска приложений на JVM-языках (Groovy, Kotlin, Scala, и др). Будут рассмотрены некоторые базовые пакеты ввода-вывода, позволяющие манипулировать данными за пределами программы. В результате прохождения курса у вас, слушатели, должно появиться знание принципов работы платформы Java, понимание того, как язык выражает принципы программирования, его объектную природу. Вы научитесь писать базовые терминальные приложения и утилиты, решать задачи (в том числе алгоритмические, не требующие подключения сложных библиотек) с использованием языка Java и с учётом его особенностей.

На этом уроке мы поговорим о

- краткой истории и причинах возникновения язка
- что нужно скачать, откуда и как это всё выбирать
- из чего всё состоит
- изучим простую структуру проекта и способы его запуска
- коротко рассмотрим утилиту джавадок

Рассмотрим примитивный инструментарий и базовые возможности платформы для разработки приложений на языке Java. В конце урока рассмотрим альтернативные способы сборки проектов, популярные и не очень.

Краткая история



Итак язык Java. Согласно википедии, Java — строго типизированный объектно-ориентированный язык программирования общего назначения, разработанный компанией Sun Microsystems. Разработка ведётся сообществом; язык и основные реализующие его технологии распространяются по лицензии GPL.
Приложения Java обычно транслируются в специальный байт-код, поэтому они могут работать на любой компьютерной архитектуре, для которой существует реализация виртуальной Java-машины. Дата официального выпуска — 23 мая 1995 года. Традиционно занимает высокие места в рейтингах популярности языков программирования (3-е место в рейтинге TIOBE (на август 2022)). На графике отлично видно, что с 2002 года язык уверенно держится в тройке популярных, значительную часть времени возглавляя его.
Но это всё - сухие факты, ничего не говорящие нам о том, чем руководствовались разработчики языка и что нам, программистам, с этим делать.

Существует, интересная версия происхождения названия языка, связанная с аллюзией на кофе машину как пример бытового устройства, для программирования которого изначально язык создавался. Кстати, в результате работы проекта мир увидел принципиально новое устройство, карманный персональный компьютер, который опередил своё время более чем на 10 лет, но из-за большой стоимости не смог произвести переворот в мире технологии и был забыт. Устройство Star7 не пользовалось популярностью, в отличие от языка программирования Java и его окружения. С середины 1990-х годов язык стал широко использоваться для написания клиентских приложений и серверного программного обеспечения. Тогда же некоторое распространение получила технология Java-апплетов — графических Java-приложений, встраиваемых в веб-страницы; с развитием возможностей динамических веб-страниц технология стала применяться редко и язык стал применяться для бэк-энда.

Почему Java?

Написано однажды, работает везде.

Итак Язык программирования Java занял свою нишу в системном и прикладном программировании из-за своей высокой скорости работы, и веб-программировании, из-за удобного сетевого окружения, ставшего основой для создания бэк-энда веб-сервисов. Java очевидно испытал влияние языков C, C++, Pascal и других, так что многое в этих языках достаточно досконально изучить один раз, чтобы иметь возможность понимать все связанные.

Поскольку язык изначально проектировался для множества разнообразных исполнителей с разными архитектурами процессоров и систем, было принято решение отделить исполнителя от решаемой прикладной задачи, так появилась виртуальная машина Java, речь о которой пойдёт немного позже. Основной смысл в том, что у языка Java появился девиз: Написано однажды, работает везде.

Задания для самопроверки

1. Как Вы думаете, почему язык программирования Java стал популярен в такие короткие сроки? (3)
   - существовавшие на тот момент Pascal и C++ были слишком сложными;
   - Java быстрее C++;
   - Однажды написанная на Java программа работает везде.

Базовый инструментарий

1. Eclipse
2. NetBeans
3. IntelliJ IDEA
4. BlueJ
5. Oracle JDeveloper
6. MyEclipse
7. Greenfoot
8. jGRASP
9. JCreator
10. DrJava

Мы привыкли к тому, что для программирования нужна некоторая среда, которая будет удобно подсказывать нам, что делать, как писать программу, запускать написанный код и помогать нам отлаживать его. Чаще всего это так, но я призываю вас не останавливаться на единственном инструменте, ведь если быть всегда сосредоточенным на отвёртке, вы никогда не узнаете, что люди придумали очень удобный шуруповёрт.
Я не планирую рекламировать ту или иную среду, скорее всего не буду даже много говорить о плюсах и минусах той или иной среды, просто покажу список и коротко расскажу о главных действующих лицах:

NetBeans - если коротко, то эта среда с нами из уважения к истории, она была первой, созданной для разработки на языке Java. Долгое время разработка этой среды не велась, но в последние несколько лет разработчики как будто бы вспомнили о ней и активно взялись за дело, сейчас есть версии под все популярные ОС, поддерживается весь популярный современный функционал, есть прикольные самобытные моменты, вроде расширенных клавиатурных макросов для написания функций с передаваемыми в них массивами;

Eclipse - проще, наверное, назвать язык или технологию, под которую нет плагина для этой весьма расширяемой и дополняемой среды программирования. Долгое время, вплоть до недавнего, а где-то и до сих пор, является корпоративным стандартом для написания Enterprise приложений и прочих сложных корпоративных приложений. Весьма гибкая среда и-за того, что разрабатывается Eclipse Foundation к ней можно подключить плагины, решающие почти любые задачи разработки. Из того что я видел своими глазами, Siemens использует Eclipse для разработки своих систем поддержки жизненного цикла изделий;

IntelliJ IDEA - стандарт де-факто примерно десять последних лет. Не уверен, можно ли назвать эту среду российской разработкой, но у компании, разрабатывающей эту среду, три русских учредителя. Да и сама эта компания началась именно с этой среды. В этой среде можно настроить почти любую деталь, что часто бывает очень удобно. Среда расширяется плагинами на почти все случаи жизни разработчика. Часто входит в экосистему JetBrains, развёрнутую на предприятии, поэтому тесно интегрируется в другие сферы деятельности компании. Существует как бесплатная, так и платная версия с расширенным функционалом, например, доступа к базам данных.

Отдельно стоит обговорить такую среду, как Android Studio, как мог заметить внимательный зритель, её нет в списке на слайде, но не потому, что в ней нельзя разрабатывать Java-приложения, а потому что если не разрабатывать в ней именно Android приложения, следует отказаться от её использования из-за некоторой избыточности. Среда снабжена значительным количеством надстроек и эмуляторов мобильных устройств, которые попросту будут вам мешать, если вы не используете их в своей профессиональной деятельности. С другой стороны, если ваша цель - это именно мобильная разработка под Android, то и выбора у вас особо нет, только Android Studio.

Задания для самопроверки

1. Как Вы думаете, почему среда разработки IntelliJ IDEA стала стандартом де-факто в коммерческой разработке приложений на Java? (4)
   - NetBeans перестали поддерживать;
   - Eclipse слишком медленный и тяжеловесный;
   - IDEA оказалась самой дружелюбной к начинающему программисту;
   - Все варианты верны.

Что нужно скачать

1. ждк
2. среду

Для разработки на языке Java вам может понадобиться довольно много разного инструментария, тем более, что требования к инструментарию отличаются ещё и от команды к команде, вам может потребоваться средство для работы с БД, средства моделирования и описания систем, дополнительные средства документирования кода и решений. Но совершенно точно можно сказать, что без двух вещей обойтись не удастся: это иде, они же среды программирования и ждк, он же инструментарий разработчика на java. О средах программирования дополнительно только что поговорили, нужно выбрать какую-нибудь, и скачать. Скорее всего вы это сделали для прохождения более ранних курсов, сейчас сможете для себя выбрать инструмент более осознанно. В курсе, кроме этой лекции и следующего за ней семинара, будет использоваться комьюнити версия среды Intellij IDEA, можете для простоты выбрать её, что вы уже скорее всего и сделали. Наверняка, для прохождения предыдущих курсов вы уже скачали и установили JDK какой-нибудь хорошей новой версии, а сейчас пришла пора разобраться, что это, какие они бывают, откуда их брать и как их выбирать, мы же за осознанность как-никак.

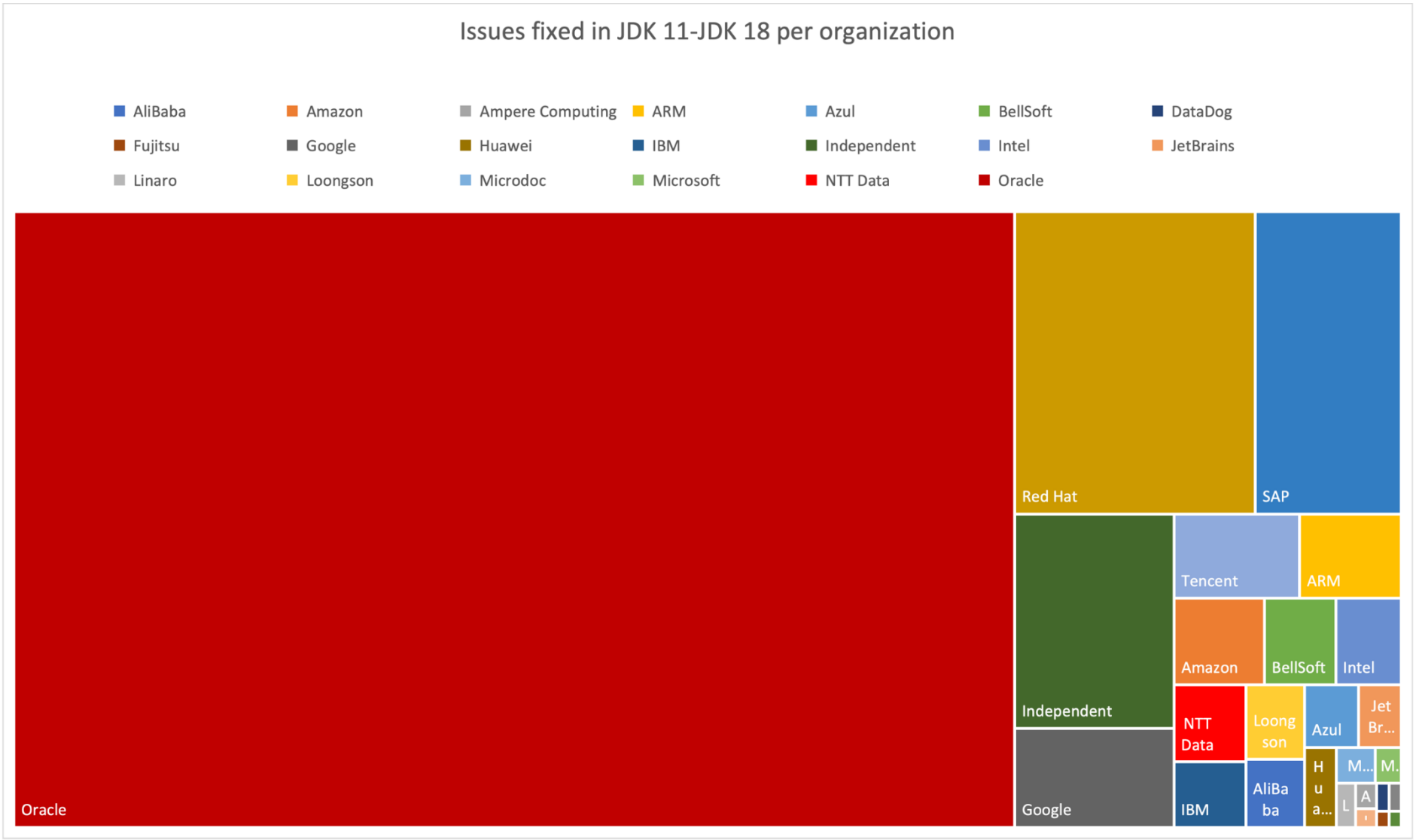
Что нужно скачать

1. Oracle JDK
2. OpenJDK by Oracle
3. Liberica JDK
4. экзотические
   - GOST Java
   - AdoptOpenJDK
   - Red Hat OpenJDK
   - Azul Zulu
   - Amazon Corretto

Итак, вендоры и версии. JDK развивается, поэтому выходят новые версии, исправляющие предыдущие недочёты, добавляющие новую функциональность. Самая распространённая версия в современной российской действительности - восьмая, но есть очень серьёзный тренд на переход к 11 и 13й версиям, на более новые часто смотрят с опаской. Распространённых вендоров три - оракл, опен, либерика. Есть, конечно, компании, собирающие свой инструментарий из открытых источников самостоятельно, но их крайне мало. Про экзотических вендоров рассказывать смысла не много, важно помнить, что они есть и у них у всех разные лицензионные политики. Я в курсе буду использовать либерику, как делаю последние несколько лет, хотя довольно долгое время пользовался ораклом. Поскольку я работаю в окологосударственной компании, не могу не упомянуть о ГОСТ джаве, которая часто всерьёз рассматривается в организациях, подобных моей, и об опенЖДК, который является основой для самостоятельных сборок инструментария.

Наиболее популярной, конечно, является оракл ждк, но, если мне будет позволено высказать персональное мнение, это больше дань традиции, нежели какая-то жизненная необходимость.

Кто делает для нас JDK



Если внимательно посмотреть на компании, внесшие вклад в развитие открытой части инструментария, начиная с ждк11 и до актуального на сегодня ждк18, конечно, максимально вкладывается оракл, но и беллсофт, производитель либерики, там будет в первой десятке. Повторюсь, я на курсе буду использовать либерику ждк11, благо для неё есть и докер-образ, если нужно как-то автоматизировать сборки вашего приложения, например; вы можете использовать любую другую не старше 8. Все примеры я буду приводить с учётом именно 8й версии языка, чтобы рассмотреть подходящий подавляющему большинству людей инструментарий. Если у вас уже установлена 11я версия от другого вендора - ничего менять не нужно, если установлена более новая версия, есть вероятность, что некоторые примеры будут отмечаться как чрезвычайно устаревшие, возможно, эти предупреждения на этапе обучения следует игнорировать. Для выбора вендора внимательно ознакомьтесь с лицензией, а для выбора версии языка руководствуйтесь здравым смыслом: большинство компаний в России и мире работают на 8, 11 или 13й версиях. Зачем учиться на какой-нибудь 18-й версии, если потом на работе вы не увидите половины каких-нибудь новых фич и фактически,

Из-за обилия вендоров и версий, которые нужно поддерживать, создаётся необходимость в автоматизации переключения между версиями. На помощь приходят инструменты, наподобие SDKMan, написанные на языке bash, позволяющие осуществить этот самый рутинный и утомительный менеджмент. Очевидным недостатком SDKMan является то, что помимо самого СДКМан нужно установить требующиеся версии и варианты ЖДК специальными внутренними командами. Хотя, например, можно перевести приложение в режим офлайн и попробовать как-то поплясать с бубном вокруг настроек. Думаю, что если в команде используется этот инструмент, для него уже есть все необходимые скрипты настроек и автоматизации. Есть и другие варианты, об одном из них (Docker) мы поговорим позднее в этой лекции.

Когда нужно по-быстрому

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. Да, многие думают, что Jupyter ноутбуки - это только для языка пайтон или для скриптовых языков, но это не так. Архитектура юпитер ноутбука позволяет ему работать с любым ядром, главное, чтобы ядро умело корректно интерпретировать написанное в ячейке с кодом. Ядро IJava делает именно это - интерпретирует Java-код, используя установленный на компьютере разработчика JDK.

Переменные среды

- `PATH`
- `JAVA_HOME`
- `JRE_HOME`
- `J2SDKDIR`
- `J2REDIR`

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

- в системную `PATH` добавить путь до исполняемых файлов JDK
- `JAVA_HOME` путь до корня JDK
- `JRE_HOME` путь до файлов JRE из состава установленной JDK
- `J2SDKDIR` устаревшая переменная для JDK, используется некоторыми старыми приложениями
- `J2REDIR` устаревшая переменная для JRE, используется некоторыми старыми приложениями

Это позволит вам не особенно много думать о настройках видимости библиотек ОС при дальнейшей установке инструментария, например, сборщиков проектов.

Задания для самопроверки

1. Чем отличается SDK от JDK? (J это частный случай, больше спец. библиотек)
2. Какая версия языка (к сожалению) остаётся самой популярной в разработке на Java? (8, 1.8)
3. Какие ещё JVM языки существуют? (Scala, Kotlin, Groovy)

JDK и его друзья

TL;DR:

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

Очень часто в интернете, при скачивании какого-то программного обеспечения, связанного с Java можно увидеть разные аббревиатуры: JRE, JVM, JDK и тому подобные, и чтобы в них не запутаться давайте коротко и быстро разберёмся что они значат, потому что я чувствую, как вам уже начинают надоедать все эти вступления.

Пойдём от самого масштабного и общего к самому маленькому и частному. Самая масштабная и общая аббревиатура - это JDK Java Development Kit - инструментарий разработчика на языке Java. Это обычный SDK который содержит в себе также всё необходимое для разработки именно на языке Java здесь есть куча других аббревиатур, компилятор, средства развёртывания, создания документации, итд они позволяют удобно писать программы на языке Java и абстрагируют программиста от необходимости описывать все вспомогательные процессы непосредственно на языке программирования то есть, грубо говоря это некие классы которые инкапсулирует сложное преобразование текста программ в простой интерфейс разработчика.

Всё, что разрабатывает разработчик, исполняется в специальной среде это среда является частью JDK и позволяет исполнять все программы написанные разработчиками на конечном в компьютере. Именно эта среда и устанавливается на компьютере пользователя чтобы пользователь мог запускать приложения, написанные на языке Java. Итак, самая необходимая вещь для любого пользователя это JRE. Это аббревиатура расшифровывается как Java Runtime Environment, то есть среда исполнения программ на языке Java. Она содержит в себе основной внутренний механизм который называется Java virtual machine и библиотеку из разных классов чтобы эта виртуальная машина хорошо работала умела показывать пользователю строки осуществлять ввод-вывод фреймворк коллекций и прочее, тут тоже есть куча интересных аббревиатур, которые мы так или иначе изучим, например, Java native interface, Java database connectivity и другие. Если копнуть ещё чуть глубже, можно и JRE разделить на JSE и JEE, но нас это деление пока не очень интересует, оно произойдёт естественным образом на более старших курсах. Пока что всё, что мы будем изучать, это JSE.

JVM и что в нём происходит

рис 1.2 (стр 4) конспекта

Инструментарий разработчика мы будем так или иначе рассматривать весь оставшийся курс, части среды исполнения мы тоже будем всесторонне изучать на последующих уроках. Сейчас хотелось бы подробнее остановиться на виртуальной машине Java, поскольку понимание того, как она устроена, должно значительно облегчить для вас понимание процесса исполнения программы и, как следствие, процесс разработки этих самых программ. Виртуальная машина Java осуществляет загрузку классов программы в оперативную память, причём здесь имеется ввиду не оперативная память как аппаратная часть компьютера, а некая выделенная часть этой оперативной памяти, которой с нами поделилось операционная система. Также осуществляется управление памятью, а именно очистка и сборка мусора и непосредственное исполнение классов нашего приложения, путём компиляции методов из промежуточного байткода в непосредственные вызовы операционной системы или другого исполнителя, то есть грубо говоря преобразования классов Java в ассемблерный код конкретного компьютера. Этот процесс называется JIT компиляция.

Существуют разные Реализация виртуальных машин (Список можно посмотреть, например, в википедии), даже экзотические, например мультиязыковой интерпретатор GraalVM. По большей части виртуальные машины отличаются как раз этой частью, с Just In Time компиляцией, то есть с преобразованием методов в непосредственные машинные вызовы в реальном времени. Чем быстрее происходит эта JIT компиляция, тем, соответственно, быстрее работает приложение на виртуальной машине. JVM самостоятельно осуществляет сборку так называемого мусора, что значительно облегчает работу программиста по отслеживанию утечек памяти, но, на мой взгляд, не способствует должной концентрации внимания программиста на этом вопросе. Важно помнить, что в Java утечки памяти всё равно существуют, особенно при программировании многопоточных приложений.

Задания для самопроверки

1. JVM и JRE - это одно и тоже? (нет)
2. Что входит в состав JDK, но не входят в состав JRE? (компилятор, средства для доков, отладки, развёртывания)
3. Утечки памяти (2)
   - Невозможны, поскольку работает сборщик мусора;
   - Возможны;
   - Существуют только в C++ и других языках с открытым менеджментом памяти.

Структура проекта

- простейшие (один файл)
- обычные (несколько пакетов)
- шаблонные (формируются сборщиками)
- скриптовые (jupyter notebook)

Ну, наконец-то проект. Проекты бывают разной сложности, сейчас поговорим о проектах без применения сборщиков, потому что сборщикам и соответствующим проектам у нас будет посвящено отдельное занятие. Все проекты по сложности структуры можно разделить на четыре основные категории: простейшие, обычные, шаблонные, скриптовые.

Простейший проект

Думаю, тут надо вживую покодить, благо немного

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

```
ivan@gb src > javac Main.java
ivan@gb src > java Main
Hello, world!
```

Начнём с простейшего проекта, который будет состоять из одного файла исходников. Естественно, вы спросите, что за чушь, мы же уже сотни раз писали «привет мир» и запускали его удобной зелёной стрелочкой в ИДЕ. Есть ответ. Мы же выясняем, что там внутри, под капотом этих удобных стрелочек и привычных процессов. Да и ради простейшего проекта запускать тяжеловесную среду программирования как-то неестественно что ли.

Jupyter Notebook

также вживую открыть юпитер ноутбук, показать, что возможно запускать код в скриптовом формате

Пока далеко не ушли от простейших проектов, думаю, целесообразно будет проговорить о скриптовых возможностях Java. Поскольку язык является не только компилируемым, но и интерпретируемым благодаря работе JVM, его возмжоно использовать как скриптовый внутри юпитер ноутбука, подключив соответствующее ядро. Интересной особенностью юпитер ноутбука является разделение фронтенда и ядра, что даёт возможность заменять эти самые ядра (в терминах юпитера они называются кернелы). скачав и установив ядро IJava мы получаем шикарную возможность делать простые наброски логики, снабжая их комментариями в формате маркдаун. На самом деле можно делать и непростые наброски, но это выходит далеко за рамки нашего курса о языке. Если будет очень интересно, призываю вас самостоятельно почитать о возможностях ядра, благо на него очень хорошая документация с примерами.

Проект на Java

- пакеты,
- классы,
- метод main,
- комментарии
- ресурсы

Как я сказал в начале этого блока, шаблонные проекты, создаваемые сборщиками мы обсудим на следующей лекции, поэтому осталось только поговорить о том, из чего состоит обычный простой проект. Итак чаще всего, это пакеты, классы, метод main, комментарии, ресурсы. Обо всём по порядку. Давайте сразу договоримся, что я сначала буду давать формальное понятие, а потом буду говорить, как его можно представить в более простом виде.

Пакеты

разделить на два слайда сначала пакет с пакетами потом стоящие стройными рядами коробки или контейнеры

Пакет - это пространство имён языка. Теоретически, как мы видели на примере простейшего проекта, класс может не находиться в пакете, но на практике такого положения дел не встретить. Например, в проекте из нескольких модулей классы без пакетов просто не будут найдены. Пакет - это некоторое место хранения и создания иерархии классов проекта. Проще всего представить пакет как папку на диске, тем более, что в файловой системе пакеты так и показываются. Но пакет - это чуть более сложная сущность. Для пакетов существуют модификаторы доступа, классы, находящиеся в одном пакете доступны друг другу даже если находятся в разных проектах (это как раз и есть отличие пакетов от папок на диске). Для нас пока достаточно, что пакеты объединяют классы по смыслу. У пакетов есть правила именования: обычно это обратное доменное имя (например, для гб.ру это будет ру.гб), название проекта, и далее уже внутренняя структура. Пакеты следует писать латинскими строчными буквами. Чтобы явно отнести класс к пакету недостаточно его просто туда положить, нужно ещё прописать в классе название пакета после специального оператора package.

Классы

разделить на два слайда сначала фото пустого школьного класса, потом

Класс - это основная единица исходного кода программы. Без класса не может начаться никакое программирование на языке Java. Даже скомпилированные файлы исходного кода имеют расширение .class. Что такое классы в терминах ООП вы уже знаете, но в Java классы - это более широкое понятие. Сама программа - это тоже класс. У классов также есть модификаторы доступа. Важно про классы - это то, что в файл с расширением джава следует класть только один класс и делать его публичным. Формально, язык не запрещает иметь в файле другие, не публичные классы, но писать так свои программы - это не толкьо дурной тон, но и весьма сбивающая с толку практика. У классов также есть правила именования. Название класса - это имя существительное в именительном падеже, написанное с заглавной буквы. Если требуется назвать класс в несколько слов, применяют так называемый upper camel case, то есть пишут слова строчными буквами, делая заглавной первую букву каждого слова.

метод мейн

разделить на два слайда псвм показать точку входа как вход в пещеру

Метод мейн - это единственный в своём роде метод, который является точкой входа в программу. Метод является публичным, и должен находиться в публичном классе, то есть он доступен для исполнения любому желающему. А желает его исполнить JVM. Метод именно с такой сигнатурой (публичный, статический, ничего не возвращающий, принимающий массив строк) - это конвенция, договорённость, об именовании. JVM при старте программы среди всех пакетов и классов будет искать именно этот метод и будет передавать ему именно массив строк в качестве параметра. При создании этого метода важно полностью повторить его сигнатуру и обязательно написать его с название со строчной буквы.

комментарии

разделить на два слайда можно как в редакции «комменты и лайки туда» и слайд с типами комментариев

С комментариями всё просто - это часть кода, которую игнорирует компилятор при преобразовании исходного кода. Комментарии бывают:

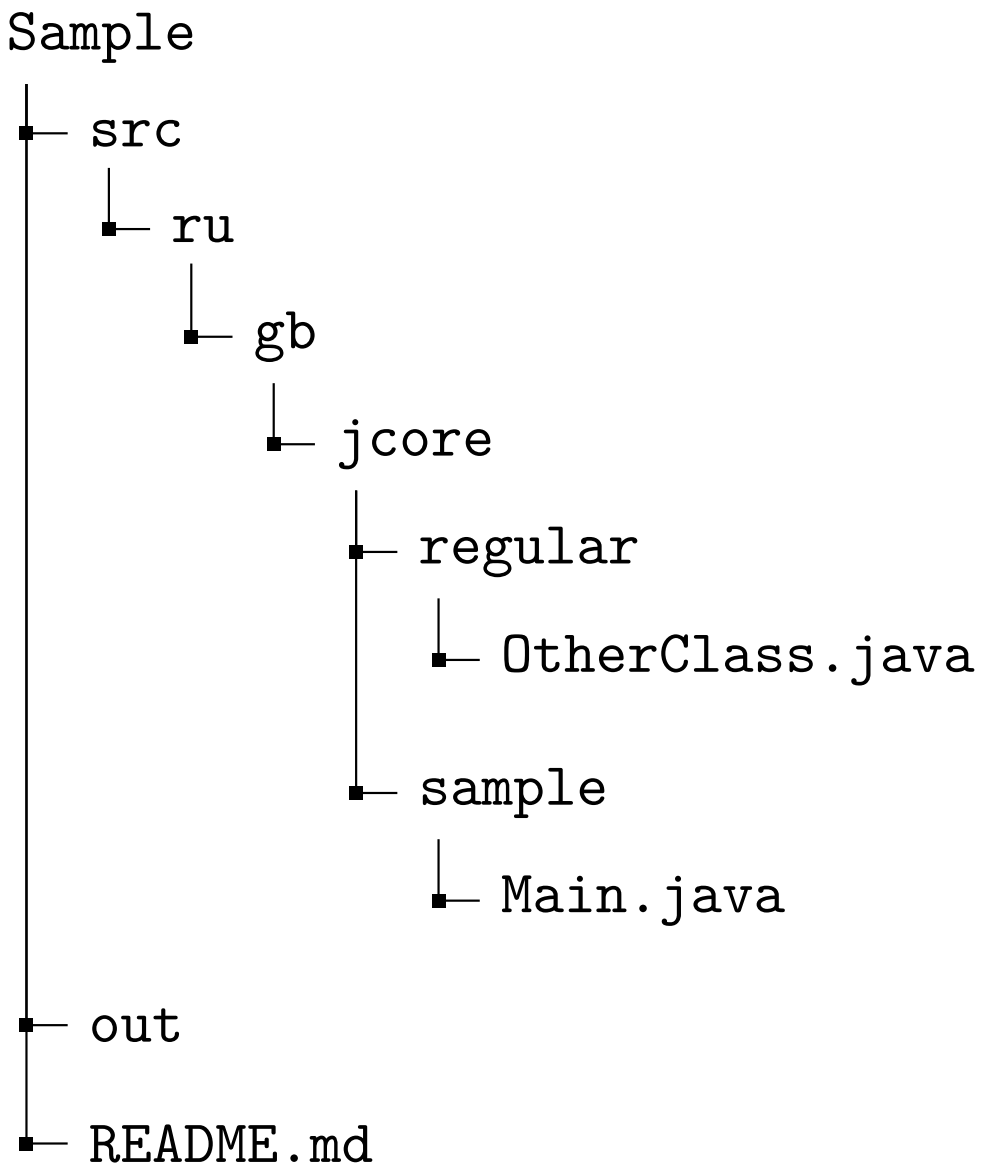
- `// comment` - до конца строки. Самый простой и самый часто используемый комментарий. Эти комментарии обычно поясняют неочевидные программные решения или сложные алгоритмы.
- `/* comment */` - внутристрочный или многострочный. Никогда не используйте его внутри строк, несмотря на то, что это возможно. Обычно эти комментарии используют, чтобы оставить какое-нибудь длинное сообщение будущим поколениям программистов, которые будут поддерживать этот код.
- `/** comment */` - комментарий-документация. Многострочный. Из него утилитой Javadoc создаётся веб-страница с комментарием. Самый полезный из всех полезных комментариев. Сейчас будем учиться делать из таких комментариев целые веб-сайты с программной документацией.

Есть ещё такое собирательное понятие, как ресурсы, но его, возможно не следует рассматривать прямо сейчас потому что оно не является обязательным для вообще любого проекта, в отличие от пакетов, классов и комментариев. Работу с ресурсами мы разберём когда будем беседовать о более сложных проектах.

Задания для самопроверки

1. Зачем складывать классы в пакеты? (структурирование, видимость*)
2. Может ли существовать класс вне пакета? (да*, нет)
3. Комментирование кода(2)
   - Нужно только если пишется большая подключаемая библиотека;
   - Хорошая привычка;
   - Захламляет исходники.

Структура простого проекта



Для дальнейших упражнений на сегодняшнем уроке нам понадобится небольшой проект, буквально из двух классов в двух пакетах, без каких-то ограничений видимости и других усложнений. На основе этого проекта мы и поговорим об оставшихся на сегодня двух несложных темах - терминальная сборка и генерация документации. Дерево проекта представлено на слайде, где папка с выходными (скомпилированными) бинарными файлами пуста, а файл README.md создан для лучшей демонстрации корня проекта.

Терминал

показать папку с простейшим проектом из одного файла, скомпилировать, запустить.

Итак. У нас был простейший проект, написать его было элементарно, что там, простой хелловорлд, скомпилировать и запустить его также просто. Нам понадобится утилита javac из состава JDK которая скомпилирует из текста программы байт-код, и утилита java из состава JRE которая этот байт-код сможет интерпретировать. То есть очевидно, и мы просто проговорим это дополнительно, что простому пользователю для запуска программы не нужен весь тяжёлый ЖДК. Показываем работу следующими командами ls /javac Main.java /ls /java Main Также существует забавный факт, что скомпилированные джава классы всегда содержат одинаковые первые 4 байта, которые в 16тиричном представлении формируют надпись «кофе, крошка», cafe babe. (открыть мейн.класс в хекс-редакторе, показать)

Терминал

тут, скорее всего, без лайв кода не обойтись, нужно открыть терминал и показать папку проекта, структуру проекта и содержимое файлов

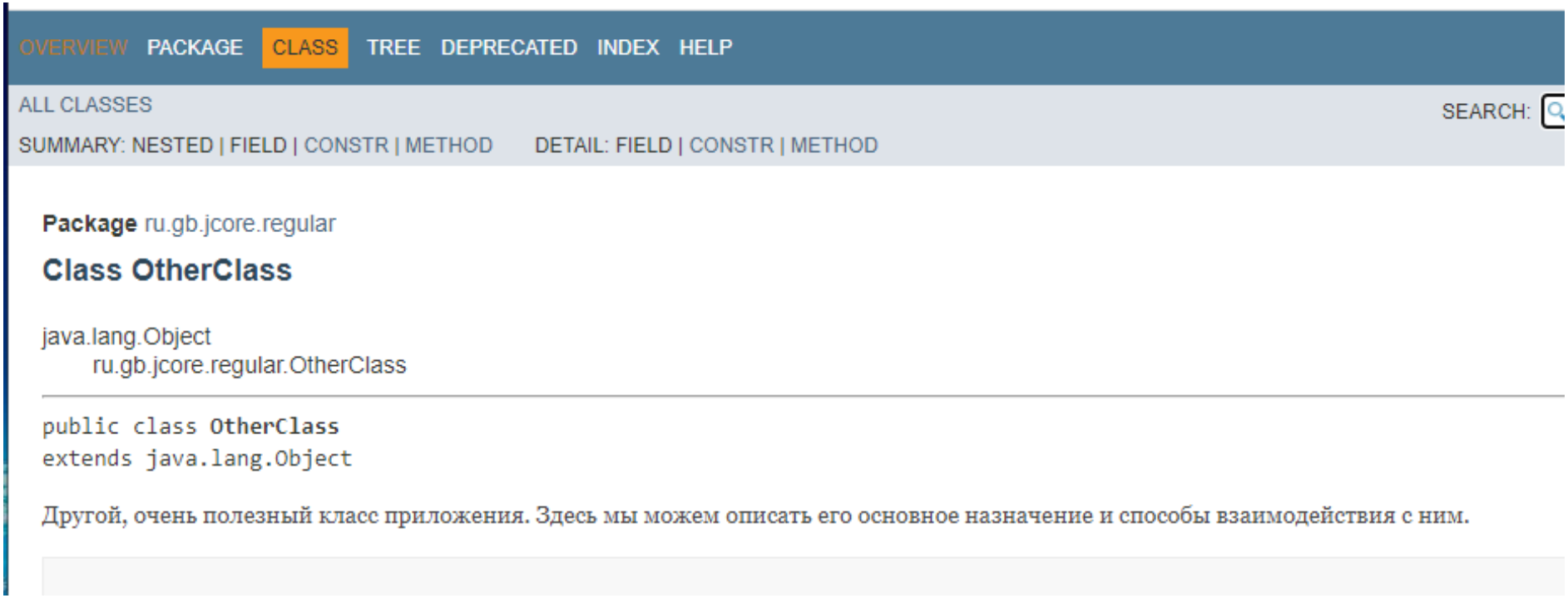
Для компиляции более сложных проектов, содержащих каике-то иерархические структуры из пакетов и классов, необходимо указать компилятору, откуда забирать файлы исходников (сорспас) и куда класть готовые классы (д может быть расшифрован как директория или как дестинейшн - назначение), а интерпретатору, откуда забирать файлы скомпилированных классов (класпас). Как хорошо видно среди только что написанных букв, для этого достаточно через пробел написать соответствующие ключи компиляции и запуска. Конечно, писать каждый раз эти ключи довольно утомительно и для автоматизации этого процесса придумали сборщики проектов, но всегда круто знать и уметь делать вещи в отсутствие сложного инструментария. Показываем работу следующими командами /javac -sourcepath ./src -d out src/ru/gb/jcore/sample/Main.java /java -classpath ./out ru.gb.jcore.sample.Main

Задания для самопроверки

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

Javadoc

Пример автоматически сгенерированной документации



OVERVIEW PACKAGE CLASS TREE DEPRECATED INDEX HELP

ALL CLASSES

SEARCH:

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Package ru.gb.jcore.regular

Class OtherClass

java.lang.Object
    ru.gb.jcore.regular.OtherClass

```
public class OtherClass
extends java.lang.Object
```

Другой, очень полезный класс приложения. Здесь мы можем описать его основное назначение и способы взаимодействия с ним.

И к последней на сегодня теме, которая мочему-то вызывает у многих даже активно практикующих программистов отторжение. К самому полезному для передачи опыта между программистами и командами программистов. К документации на свой код. Конечно, особенно ярые противники описывать свою работу сейчас возмутятся, мол, я ж программист, пусть программную документацию пишут технические писатели. Но если подумать, то техпис фактически всё равно придёт к вам, программисту, за разъяснениями, что именно делает тот или иной фрагмент кода. А комментарии к коду всё равно писать, да и техпису, скорее всего придётся отвечать в текстовом виде, так почему бы не облегчить себе жизнь сразу, документируя свой код по мере написания, например, между спринтами.

утилита javadoc

основные ключи и аргументы

- `-d docs`
- `-sourcepath src`
- `-cp out`
- `-subpackages`
- `ru`

Чтобы просто создать документацию надо вызвать утилиту javadoc с небольшим количеством ключей, давайте их разберём. Уже знакомый нам ключ -d означающий папку (или директорию, кому как больше нравится) в которую следует положить готовую документацию. Помним, ключ легко запомнить, если расшифровать его как дестинейшн, назначение. -sourcepath означает папку, где лежат исходники проекта, как мы помним, в исходниках мы пишем те самые комментарии, которые потом можно преобразовать в джавадок. -cp говорит о том, где лежат готовые скомпилированные классы, эта информация нужна, чтобы понять, как программа в итоге скомпилировалась, что в ней задействовано, что как связано и прочее. -subpackages говорит о том, что надо взять не только указанный далее пакет, но и все его подпакеты рекурсивно. Ну и, собственно, пакет, для которого нужно создать документацию, ru. Показываем работу следующими командами:
/javadoc -d docs -sourcepath src -cp out -subpackages ru

Если нужно сгенерировать документацию к конкретному пакету, например, если вы хотите разложить их по разным папкам, то надо указатеь его полный адрес, и, вероятно, убрать ключ subpackages

куда же без особенностей работы с Windows?

- `-locale ru_RU`
- `-encoding utf-8`
- `-docencoding cp1251`

Чтобы учесть то, что происходитс ОС Windows надо чуть больше, поскольку так исторически сложилось, что у Windows есть проблемки с локалью. Поскольку все программы на Java пишутя в кодировке UTF-8, а основная кодировка всех приложений, работающих под ОС виндоус это cp1251, то при создании документации стандартными средствами итоговые веб-страницы создаются на клингонском. Чтобы перевести их на земной кириллический, нужно добавить три ключа - указать какая локаль установлена -locale, в какой кодировке исходный текст -encoding utf-8, в какой кодировке должен быть итоговый документ -docencoding cp1251. Показываем работу следующими командами: /javadoc -locale ru_RU -encoding utf-8 -docencoding cp1251 -d docs -sourcepath src -cp out -subpackages ru Поскольку я сейчас не с виндоусом, то такие настройки наоборот, сломают отображение.

Задания для самопроверки

1. Javadoc находится в JDK или JRE? (ждк)
2. Что делает утилита Javadoc? (3)
   - Создаёт комментарии в коде;
   - Создаёт программную документацию;
   - Создаёт веб-страницу с документацией из комментариев.

Автоматизация сборки в CLI

makefile

Компилировать проект руками — занятие весьма утомительное, особенно когда исходных файлов становится больше одного, и для каждого из них надо каждый раз набивать команды компиляции, а ещё не забыть о том, какой файл в проекте главный, какие включены библиотеки и многое другое. Руки так и тянутся куда-то это записать, чтобы не забыть, но говорить о специальных менеджерах проектов ещё рано, поэтому научимся автоматизировать терминальную сборку. Будем учиться создавать и использовать Мейкфайлы.

Makefile — это набор инструкций для программы make (классически, это GNU Automake), которая помогает собирать программный проект буквально в одну команду. Хотя эта технология и отмирает, но все равно используется как некоторыми отдельными разработчиками, так и в проектах, особенно часто это встречается, когда в Java переквалифицируются бывшие C++ программисты. Если запустить make то программа попытается найти файл с именем по-умолчанию Makefile в текущем каталоге и выполнить инструкции из него. Эти самые мейкфайлы и сохранят для нас все нужные настройки проекта, компиляции, структуры и прочего.

Не лишним будет напомнить, что мейкфайлы, не привносят ничего принципиально нового в процесс компиляции, а только лишь автоматизируют его, поэтому важно помнить, что все действия, которые мы совершаем при ручной компиляции просто записаны в мейкфайл, и обратно, то что записано в мейкфайле может быть выполнено вручную в терминале.

В простейшем, то есть нашем, случае, в мейкфайле достаточно описать так называемую цель, таргет, и что нужно сделать для достижения этой цели. Цель, собираемая по-умолчанию называется all, так, для простейшей компиляции нам нужно написать all:

javac -sourcepath .src/ -d out src/ru/gb/jcore/sample/Main.java

Автоматизация сборки в CLI

SRCDIR := src OUTDIR := out
JC := javac JCFLAGS := -sourcepath .(SRCDIR)/ -d (OUTDIR)
MAINSOURCE := ru/gb/dj/Main MAINCLASS := ru.gb.dj.Main
all: (JC) (JCFLAGS) SRCDIR/MAINSOURCE.java
clean: rm -R (OUTDIR)
run: cd out && java MAINCLASS

По сути, это всё. Но мы же хотим, чтобы вся было гибо и не нужно было запоминать что как называется, чтобы это потом можно было поправить где-то в одном месте. Для этого в мейкфайлах придумали переменные, и они работают как самые обычные переменные, их значение подставляется в то место, где переменная вызывается. Добавив пару таргетов и немного переменных мы получим мейкфайл как на слайде, он позволит нам не только компилировать проект, но также очищать выходные файлы в случае неудачи и запускать его. Также не особенно переживая о ключах, командах и прочем.

Для того, чтобы утилита мейк сделала своё дело нужно в терминале просто её вызвать без аргументов. Чтобы воспользоваться другими написанными таргетами (автоматизациями, задачами) нужно после имени утилиты написать через пробел название таргета, например, часто пишут задачу clean, рекурсивно очищающую папку с готовыми классами, соответственно, вызов make clean как ни странно, рекурсивно удалит папку с готовыми классами.

Автоматизация сборки в CLI, контейнеризация

docker

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 почти на на любой системе, и уж точно на системах большой тройки (линукс виндоус макос).

Целью этого курса не является рассмотрение всех возможностей докер, поэтому ограничимся только теми, которые нам понадобятся, чтобы увидеть заветное, запущенное джавой, «привет мир» в терминале. Снова не лишним будет напомнить, что мы можем выполнить все те же самые действия в простом терминале. Но докер даёт нам немного преимуществ. Например, не нужно устанавливать JDK и думать о переключении между версиями, достаточно взять контейнер с нужной версией и запустить наше приложение в нём. Помните я буквально полчаса назад упоминал о том, что есть более абстрактный инструмент менеджмента версий языка и инструментария? вот, это он.

Автоматизация сборки в CLI, контейнеризация

```
# syntax=docker/dockerfile:1
FROM bellsoft/liberica-openjdk-alpine:11.0.16.1-1
# LABELS go here
# from where (related to Dockerfile) to where COPY ./src ./src
# remember that you are root and in / directory RUN mkdir ./out
# be sure to use relative paths RUN javac -sourcepath ./src -d out
./src/ru/gb/dj/Main.java
# what will we do on a container start CMD java -classpath ./out
ru.gb.dj.Main
```

С точки зрения разработки, контейнерные приложения в докере это, по сути, набор инструкций, которые вы пишете в специальном файле. Всё остальное докер берёт на себя. Он эти инструкции выполняет, виртуализует, запускает и делает все остальные свои важные вещи, а мы в результате видим заветный сервис или приложение, поднятые буквально парой команд в терминале.
Упомянутый специальный файл называется докерфайл и в нашем случае он будет мало отличаться от того, что мы писали ранее в терминале или написали в мейкфайле. Первой строкой докерфайла мы обязательно должны указать, какой виртуальный образ будет для нас основой. Тут можно выбрать пустой образ, образ ОС, например, убунту или центос, а можно выбрать сразу ЖДК, как поступим мы. FROM bellsoft/liberica-openjdk-alpine:11.0.16.1-1

Дальше скажем докеру, что надо будет при создании образа скопировать все файлы из нашей папки СРЦ внутрь образа, в папку СРЦ COPY ./src ./src

потом также при создании образа надо будет создать внутри папку аут простой терминальной командой RUN mkdir ./out
и запустить компиляцию. мы же изначально используем образ ждк, а значит там внутри есть компилятор. главное не запутаться в путях и не забывать, что команда будет выполняться в образе, а не на нашем компьютере. RUN javac -sourcepath ./src -d out ./src/ru/gb/dj/Main.java
А последняя команда говорит, что именно нужно сделать, когда контейнер создаётся из образа и заускается. CMD java -classpath ./out ru.gb.dj.Main
Как очевидно, докер-образ и как следствие докер-контейнеры возможно настроить таким образом, чтобы скомпилированные файлы складывались обратно на компьютер пользователя через общие папки, если это нужно, или, например, компилировать в разные папки классы разными версиями компилятора, и запускать их также разными версиями, имея весь этот зоопарк технологий в контейнерах, не засоряя свой компьютер.

Можно даже, например, не брать в качестве исходного образа ждк, а взять ОС, вручную установить на ней ждк и, например, мейк, и автоматизировать всё мейком внутри докера. Это странно, но, как минимум не запрещено.

итоги

что изучили и домашка

На этом уроке мы рассмотрели историю и причины создания языка, его окружение и структуру проекта, узнали, что скрывают в себе такие аббревиатуры как JDK JRE JVM SDK JIT CLI. Рассмотрели базовое применение утилит java, javac и javadoc. Даже немного посмотрели в сторону автоматизации рутинных задач.

В качестве домашнего задания попробуйте

- 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.

На следующей лекции мы поговорим о том как осуществлять управление более сложными проектами: Jar-файлы; сборщики Gradle/Maven; чтоб такое репозитории и как ими пользоваться.