# Техническая специализация Java

(1. Java Core)

Иван Игоревич Овчинников

2022-08-29 (22:43)

## Содержание

# 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.

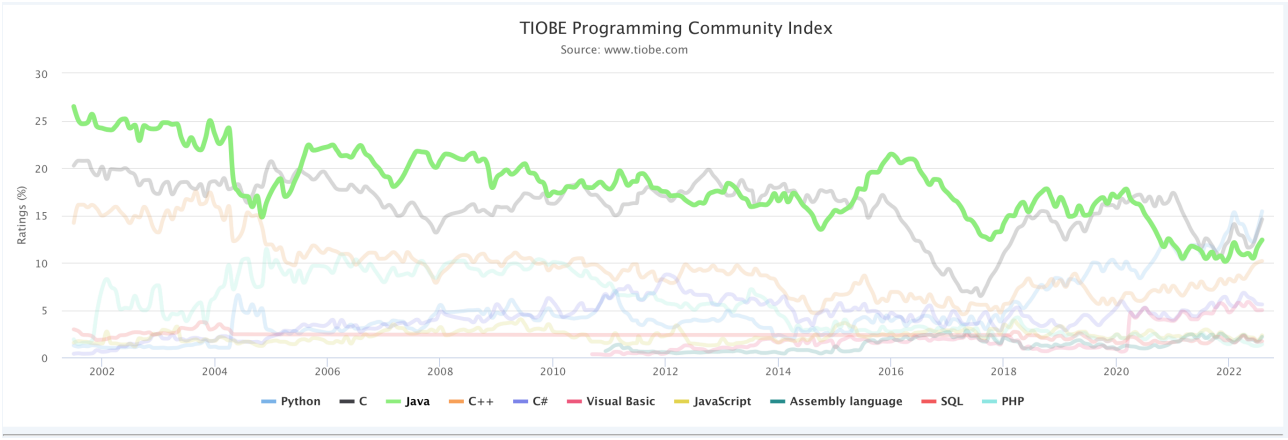



Рис. 1: График популярности языков программирования TIOBE

### 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 ин-

терфейсы, которые задействует операционная система, предоставляя в свою очередь интерфейс 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

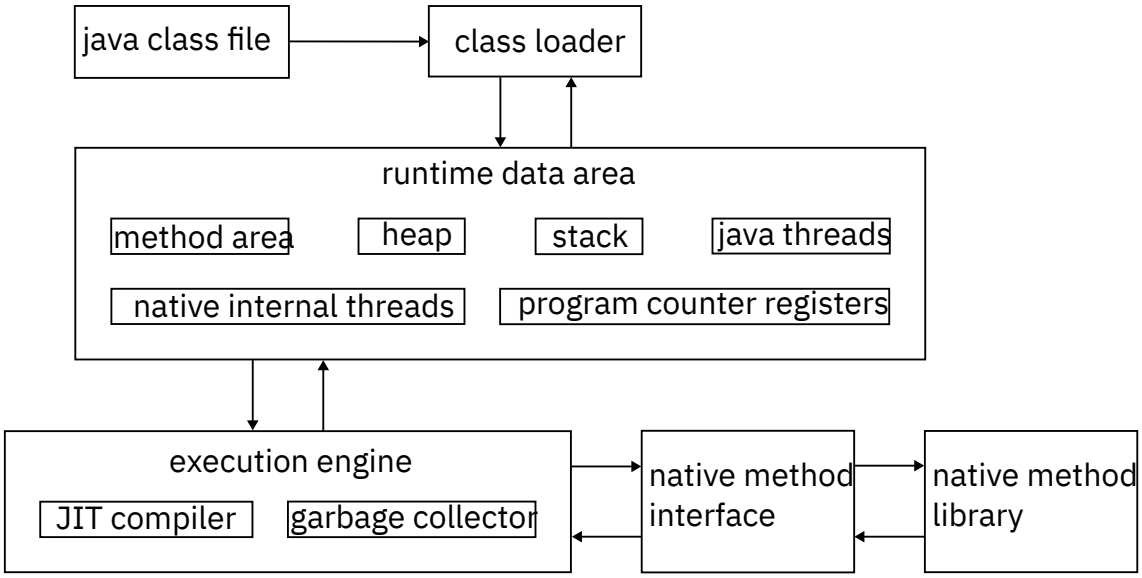



Рис. 2: принцип работы JVM

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 7, 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.

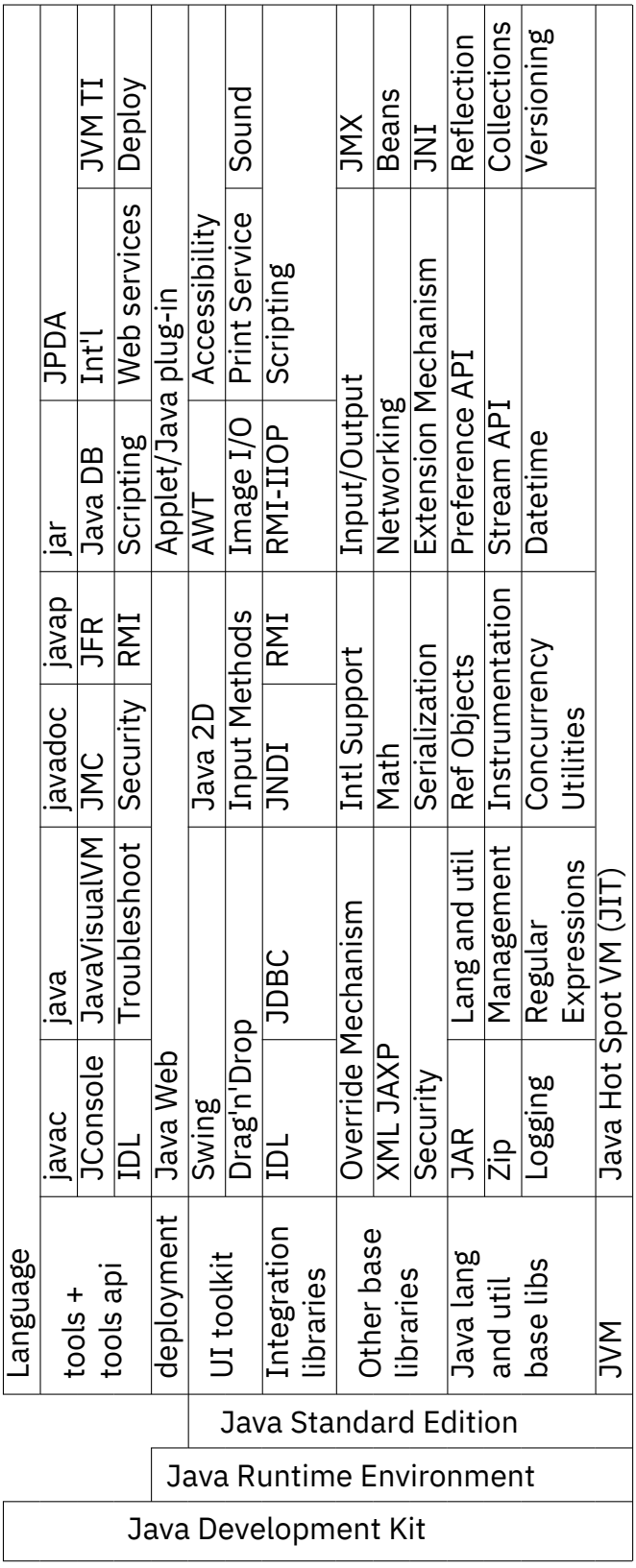



Таблица 1: Общее представление состава JDK

### 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

```
1 public class Main {
2    public static void main(String[] args) {
3       System.out.println("Hello, world!");
4    }
5 }
```

**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.

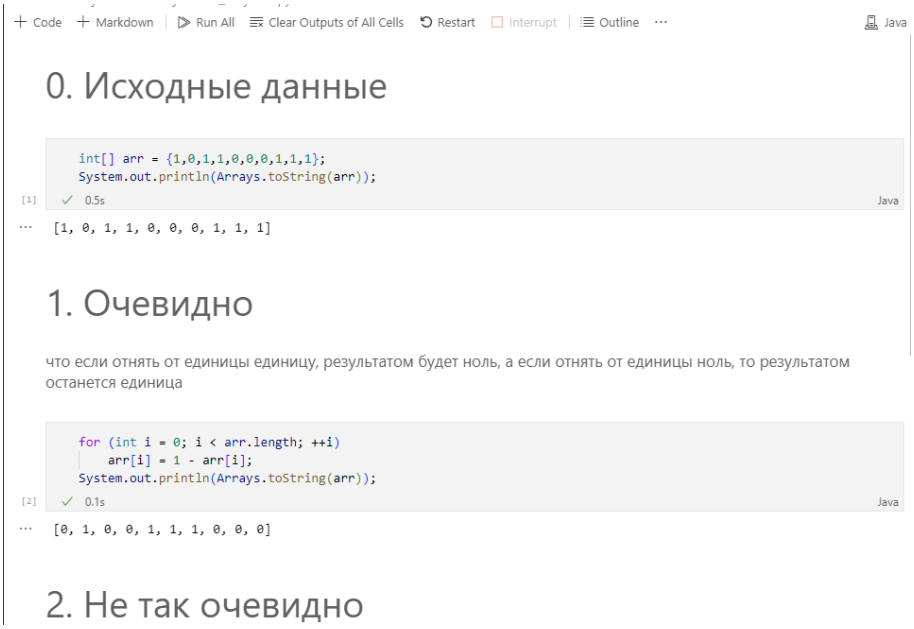



Рис. 3: Пример простого Java проекта в Jupyter Notebook

**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 прави-

ла 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.

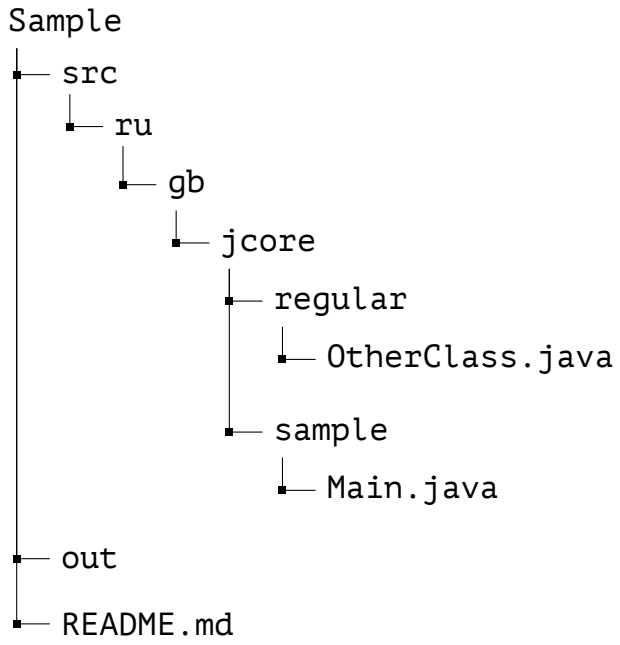



Рис. 4: Структура простого проекта

Содержимое файлов исходного кода представлено ниже.

```java
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));
   }
}
```

```java
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.

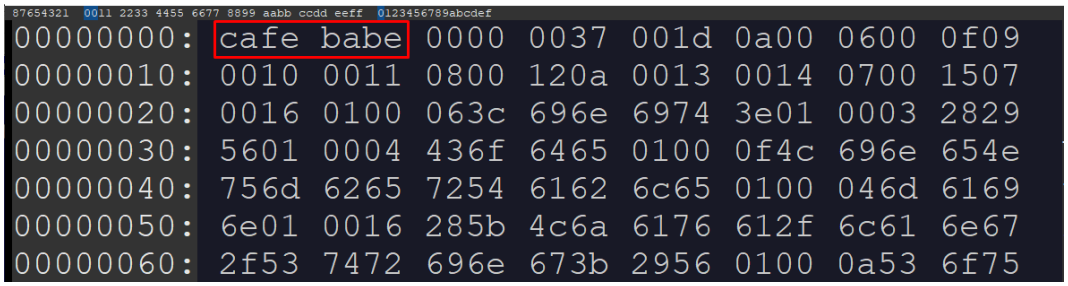
## 1.7. Отложим мышки в сторону (CLI: сборка, пакеты, запуск)

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».




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.

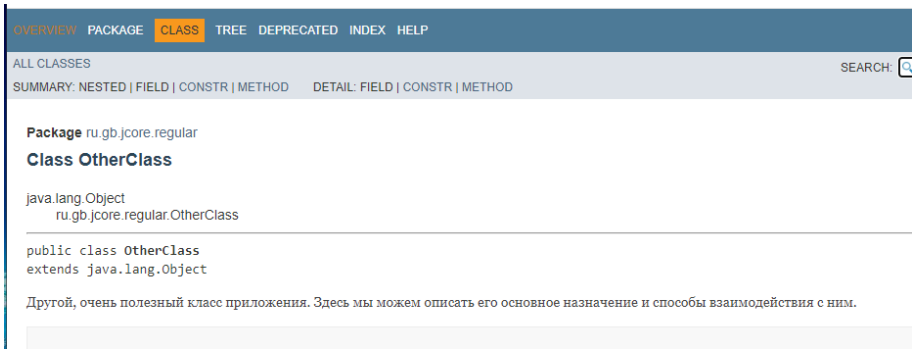



Рис. 5: Часть страницы автосгенерированной документации

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:

```
all:
    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:

```
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.

# 2. Управление проектом: сборщики проектов

## 2.1. В предыдущих сериях...

В прошлом разделе мы рассмотрели:
— краткую историю;
— что скачать, и как это выбирать;
— какие шестерёнки крутятся внутри;
— структуру простого и обычного проекта;
— стандартную утилиту для создания документации;
— сторонние инструменты автоматизации сборки.

## 2.2. В этом разделе

Будет коротко рассмотрена мотивация создания и использования сборщиков проектов (зачем это нужно), какой эволюционный путь прошли специализированные сборщики проектов, какие новые понятия появились при их использовании. Рассмотрим два самых популярных на сегодняшний день сборщика и один экзотический, но достаточно быстро набирающий обороты. Поймём какой путь проделывает чужая библиотека, чтобы попасть в наш проект и научимся публиковать собственный код, делая тем самым вклад в сообщество.

— Ant
— Ivy;
— Maven;
— Gradle;
— Bazel;
— Артефакт;
— Репозиторий;
— Прокси;

## 2.3. Мотивация и схема (зачем это нужно и как это работает)

Сборщие проекта - это сложный инструментарий, который может скрывать обширный объем работы, который при ручном подходе требует значительных затрат и может оказаться весьма утомительным.

Позвляют не привязываться к конкретным IDE и интерфейсу среды разработки. Имеют текстовое управление, что часто ускоряет процесс работы с ними на пред-продакшн серверах, где часто терминальный интерфейс.

> Система сборки это программа, которая собирает другие программы. На вход система сборки получает исходный код, написанный разработчиком, а на выход выдаёт программу, которую уже можно запустить. Отличается от компилятора тем, что вызывает его при своей работе, а компилятор о системе сборки ничего не знает, то есть является инструментом более широкого спектра.

Сборщик часто решает целый спектр задач, для решения которых компилятор не предусмотрен. Например:

— загрузить зависимые библиотеки;
— скомпилировать классы модуля;
— сгенерировать дополнительные файлы: SQL-скрипты, XML-конфиги и пр.;
— упаковать скомпилированные классоы в архивы;
— компилировать и запустить модульные тесты и рассчитать процент покрытия;
— развернуть (deploy) файлы проекта на удаленный сервер;
— генерировать документацию и отчеты.

Системы сборок имеют схожую верхнеуровневую архитектуру:

1. Конфигурации - собственная конфигурация, где хранятся «личные» настройки системы. Например, такие как информация о месте установки или окружении, информация о репозиториях и прочее;
2. Конфигурация модуля, где описывается место расположения проекта, его зависимости и задачи, которые требуется выполнять для проекта;
3. Парсеры конфигураций - парсер способный «прочитать» конфигурацию самой системы, для её настройки соответствующим образом;
4. Парсер конфигурации модуля, где некоторыми «понятными человеку» терминами описываются задачи для системы сборки;
5. Сама система — некоторая утилита + скрипт для её запуска в вашей ОС, которая после чтения всех конфигураций начнет выполнять тот или иной алгоритм, необходимый для реализации запущенной задачи;
6. Система плагинов — дополнительные подключаемые надстройки для системы, в которых описаны алгоритмы реализации типовых задач;
7. Локальный репозиторий — репозиторий (некоторое структурированное хранилище некоторых данных), расположенный на локальной машине, для кэширования запрашиваемых файлов на удаленных репозиториях.

Для языка Java основных систем сборок три:

— Иногда можно встретить Ant в сочетании с инструментом по управлению зависимостями Ivy, в чистом виде Ant почти никогда не применялся;
— Самый популярный инструмент - это Maven, используется для JavaEE и приложений с использованием Spring Framework;
— Gradle основной инструмент в мобильной разработке, часто Groovy/Kotlin описания сборки оказываются удобнее XML;

Экзотическая Bazel. Её отличает полная кроссплатформенность и кросстехнологичность, что делает её особенно привлекательной для микросервисных проектах, использующих

несколько языков программирования.

## 2.4. С чего всё начиналось (Ant, Ivy)

Первый специализированный инструмент автоматизации задач для языка Java. По сути, это аналог `Makefile`, то есть набор скриптов (которые в разговорной речи называются тасками[2]). В отличие от make, утилита Ant имеет только одну зависимость — JRE. Ещё одним важным отличием от мейка является отказ от использования команд операционной системы. Всё что пишется в ант пишется на XML, что обеспечивает переносимость сценариев между платформами.

Управление процессом сборки как только что было сказано, происходит посредством XML-сценария, также называемого Build-файлом. В первую очередь этот файл содержит определение проекта, состоящего из отдельных целей (таргетов, с этим понятием мы познакомились когда говорили о мейкфайлах). Цели в терминах ант сравнимы с процедурами в языках программирования и содержат вызовы команд-заданий (тасков). Каждое задание представляет собой неделимую, атомарную команду, выполняющую некоторое элементарное действие.

Между целями могут быть определены зависимости — каждая цель выполняется только после того, как выполнены все цели, от которых она зависит (если они уже были выполнены ранее, повторного выполнения не производится). Типичными примерами целей являются clean (удаление промежуточных файлов), compile (компиляция всех классов), deploy (развёртывание приложения на сервере).

Скачивание и установка, ant для Linux-подобных ОС выполняется привычной командой установки в вашем пакетном менеджере, вроде sudo apt install ant, а для Windows можно перейти на сайте ant.apache.org и скачать архив, распаковав его и прописав соответствующий путь в переменные среды. Проверить версию можно командой ant -version

```
<?xml version="1.0"?>
<project name="HelloWorld" default="hello">
  <target name="hello">
    <echo>Hello, World!</echo>
  </target>
</project>
```

Теперь можно написать простой сценарий HelloWorld: в нём мы указываем версию хмл, без этого никакой хмл никогда не работает. затем говорим, что наш проект называется приветмир, а собирать по умолчанию для этого проекта надо таргет хелло. далее в проекте идут описания таргетов, в нашем случае таргет будет один, он будет носить имя хелло, а в рамках таргета нужно будет вывести в терминал приветствие миру.

```
mkdir hello
cd hello
ant
```

[2] от англ task - задача, задание

Затем, нужно создать подкаталог hello и сохранить туда файл с именем build.xml, который содержит сценарий. Далее надо перейти в каталог и вызвать ant. Полный перечень команд:

```
Buildfile: /home/hello/build.xml

hello:
[echo] Hello, World!
BUILD SUCCESSFULL
```

В результате выполнения которых получим следующий вывод. Что именно произошло? система сборки нашла файл сценария с именем по умолчанию, по удачному стечению обстоятельств система ищет файл с именем build.xмл и выполнила цель указанную по умолчанию, здесь уже никаких удачных стечений обстоятельств, мы явно указали, что нужно по умолчанию выполнять таргет с именем hello. Чтобы не запутаться в более сложном проекте, мы указали имя проекта, с помощью атрибута name.

```
<!-- Очистка -->
<target name="clean" description="Removes all temporary files">
  <!-- Удаление файлов -->
  <delete dir="${build.classes}"/>
</target>
```

В стандартной поставке ant присутствует более 100 заранее созданных заданий, таких как: удаление файлов и директорий (delete), компиляция java-кода (javac), вывод сообщений в консоль (echo) и т.д. Вот пример реализации удаления временных файлов, используя задание delete

## 2.5. Репозитории, артефакты, конфигурации

## 2.6. Классический подход (Maven)

## 2.7. Всем давно надоел XML (Gradle)

## 2.8. Собственные прокси, хостинг и закрытая сеть

## 2.9. Немного экзотики (Bazel)

Ant, Ant+Ivy Первым для автоматизации этих задач появился Ant. Это аналог make-файла, а по сути набор скриптов (которые называются tasks). В отличие от make, утилита Ant полностью независима от платформы, требуется лишь наличие на применяемой системе установленной рабочей среды Java — JRE. Отказ от использования команд операционной системы и формат XML обеспечивают переносимость сценариев. Управление процессом сборки происходит посредством XML-сценария, также называемого Build-файлом. В первую очередь этот файл содержит определение проекта, состоящего из отдельных целей (Targets). Цели сравнимы с процедурами в языках программирования и содержат вызовы команд-заданий (Tasks). Каждое задание представляет собой неделимую, атомарную команду, выполняющую некоторое элементарное действие. Между целями могут быть определены за-

висимости — каждая цель выполняется только после того, как выполнены все цели, от которых она зависит (если они уже были выполнены ранее, повторного выполнения не производится). Типичными примерами целей являются clean (удаление промежуточных файлов), compile (компиляция всех классов), deploy (развёртывание приложения на сервере). Скачивание и установка, в случае каких-то неисправностей, ant для Linux выполняется командой вроде sudo apt-get install ant, а для Windows можно перейти на сайте ant.apache.org и скачать архив. Проверить версию можно командой ant -version Теперь можно написать простой сценарий HelloWorld

<?xml version="1.0"?> <project name="HelloWorld"default="hello» <target name="hello» <echo>
World!</echo> </target> </project>

Затем, нужно создать подкаталог hello и сохранить туда файл с именем build.xml, который содержит сценарий. Далее надо перейти в каталог и вызвать ant. Полный перечень команд:

```
  $ mkdir hello
  $ cd hello
  $ ant
  Buildfile: /home/hello/build.xml

hello:
[echo] Hello, World!
BUILD SUCCESSFULL
```

Теперь нужно бы пояснить, что произошло: система сборки нашла файл сценария с указанным по умолчанию именем build и выполнила цель с именем hello, который указан в теге проекта, с помощью атрибута default со значением hello, а также имя проекта, с помощью атрибута name. В стандартной версии ant присутствует более 100 заданий, такие как: удаление файлов и директорий (delete), компиляция java-кода (javac), вывод сообщений в консоль (echo) и т.д. Вот пример реализации удаления временных файлов, используя задание delete:

```
  <!-- Очистка -->
  <target name="clean" description="Removes all temporary files">
  <!-- Удаление файлов -->
  <delete dir="${build.classes}"/>
  </target>
```

На данный момент Ant используют в связке с Ivy, которая является гибким, настраиваемым инструментом для управления (записи, отслеживания, разрешения и отчетности) зависимостями Java проекта. Некоторые достоинства Ivy:

гибкость и конфигурируемость – Ivy по существу не зависит от процесса и не привязан к какой-либо методологии или структуре; тесная интеграция с Apache Ant – Ivy доступна как автономный инструмент, однако он особенно хорошо работает с Apache Ant, предоставляя ряд мощных задач от разрешения до создания отчетов и публикации зависимостей; транзитивность – возможность использовать зависимости других зависимостей.

Немного о функциях Ivy:

управление проектными зависимостями; отчеты о зависимостях. Ivy производит два основных типа отчетов: отчеты HTML и графические отчеты; Ivy наиболее часто используется для разрешения зависимостей и копирует их в директории проекта. После копирования зависимостей, сборка больше не зависит от Apache Ivy; расширяемость. Если настроек по умолчанию недостаточно, вы можете расширить конфигурацию для решения вашей проблемы. Например, вы можете подключить свой собственный репозиторий, свой собственный менеджер конфликтов; XML-управляемая декларация зависимостей проекта и хранилищ JAR; настраиваемые определения состояний проекта, которые позволяют определить несколько наборов зависимостей.

Apache Ivy обязана быть сконфигурирована, чтобы выполнять поставленные задачи. Конфигурация задается специальным файлом, в котором содержится информация об организации, модуле, ревизии, имени артефакта и его расширении. Некоторые модули могут использоваться по разному и эти различные пути использования могут требовать своих, конкретных артефактов для работы программы. Кроме того, модуль может требовать одни зависимости во время компиляции и сборки, и другие во время выполнения. Конфигурация модуля определяется в Ivy файле в формате .xml, он будет использоваться, чтобы обнаружить все зависимости для дальнейшей загрузки артефактов. Понятие «загрузка» артефакта имеет различные варианты выполнения, в зависимости от расположения артефакта: артефакт может находиться на веб-сайте или в локальной файловой системе вашего компьютера. В целом, загрузкой считается передача файла из хранилища в кеш Ivy. Пример конфигурации зависимостей с библиотекой Lombok:

<ivy-module version="2.0» <info organisation="org.apache"module="hello-ivy"/> <dependencies>
<dependency org="org.projectlombok"name="lombok"rev="1.18.24"conf="build->master"/> </depe
</ivy-module>

Его структура довольно проста, первый корневой элемент <ivy-module version="2.0» указывает на версию Ivy, с которой данный файл совместим. Следующий тег <info organisation="o
ivy"/> содержит информацию о программном модуле, для которого указаны зависимости, здесь определяются название организации разработчика и название модуля, хотя можно написать в данный тег что угодно. Наконец, сегмент <dependencies> позволяет определить зависимости. В данной примере модулю необходимо одна сторонняя библиотека: lombok. Однако, у такой системы, как Ant есть и свои минусы: Ant-файлы могут разрастаться до нескольких десятков мегабайт по мере увеличения проекта. На маленьких проектах выглядит всё достаточно неплохо, но на больших они длинные и неструктурированные, а потому сложны для понимания. Что привело к появлению новой системы - Maven.

## 3. Специализация: данные и функции

Базовые функции языка: математические операторы, условия, циклы, бинарные операторы; Данные: типы, преобразование типов, константы и переменные (примитивные, ссылочные), бинарное представление, массивы (ссылочная природа массивов, индексация, манипуляция данными); Функции: параметры, возвращаемые значения, перегрузка функций;

## 3.1. Данные

Хранение данных в Java осуществляется привычным для программиста образом: в переменных и константах. Языки программирования бывают типизированными и нетипизированными (бестиповыми).

Отсутствие типизации в основном присуще старым и низкоуровневым языкам программирования, например, Forth, некоторые ассемблеры. Все данные в таких языках считаются цепочками бит произвольной длины и, как следует из названия, не делятся на типы. Работа с ними часто труднее, и при чтении кода не всегда ясно, о каком типе переменной идет речь. При этом часто безтиповые языки работают быстрее типизированных, но описывать с их помощью большие проекты со сложными взаимосвязями довольно утомительно.

> Java является языком со строгой (сильной) явной статической типизацией.

Что это значит?

— Статическая - у каждой переменной должен быть тип и мы этот тип поменять не можем. Этому свойству противопоставляется динамическая типизация;
— Явная - при создании переменной мы должны ей обязательно присвоить какой-то тип, явно написав это в коде. Бывают языки с неявной типизацией, например, Python;
— Строгая(сильная) - невозможно смешивать разнотипные данные. С другой стороны, существует JavaScript, в котором запись `2 + true` выдаст результат `3`.

Все данные в Java делятся на две основные категории: примитивные и ссылочные.

Данные: типы, преобразование типов, константы и переменные (примитивные, ссылочные), бинарное представление, массивы (ссылочная природа массивов, индексация, манипуляция данными);

| Тип | Пояснение | Диапазон |
|---|---|---|
| byte | Самый маленький из адресуемых типов, 8 бит, знаковый | [−128, +127] |
| short | Тип короткого целого числа, 16 бит, знаковый | [−32 768, +32 767] |
| char | Целочисленный тип для хранения символов в кодировке UTF-8, 16 бит, беззнаковый | [0, +65 535] |
| int | Основной тип целого числа, 32 бита, знаковый | [−2 147 483 648, +2 147 483 647] |
| long | Тип длинного целого числа, 64 бита, знаковый | [−9 223 372 036 854 775 808, +9 223 372 036 854 775 807] |
| float | Тип вещественного числа с плавающей запятой (одинарной точности, 32 бита) | |
| double | Тип вещественного числа с плавающей запятой (двойной точности, 64 бита) | |
| boolean | Логический тип данных | true, false |

Рис. 6: Основные типы данных в языке C

Базовые функции языка: математические операторы, условия, циклы, бинарные операторы;

Функции: параметры, возвращаемые значения, перегрузка функций;

### 3.1.1. Антипаттерн "магические числа"

В прошлом примере мы использовали антипаттерн - плохой стиль для написания кода. Число 18 используется в коде коде без пояснений. Такой антипаттерн называется "магическое число". Рекомендуется помещать числа в константы, которые храняться в начале файла. ADULT = 18 age = float(input('Ваш возраст: ')) how_old = age - ADULT print(how_old, "лет назад ты стал совершеннолетним")

Плюсом такого подхода является возможность легко корректировать большие проекты. Представьте, что в вашем коде несколько тысяч строк, а число 18 использовалось несколько десятков раз.  При развертывании проекта в стране, где совершеннолетием считается 21 год вы будете перечитывать весь код в поисках магических "18"и править их на "21". В случае с константой изменить число нужно в одном месте.  Дополнительный сложности могут возникнуть, если в коде будет 18 как возраст совершеннолетия и 18 как коэффициент для рассчёт чего-либо. Теперь править кода ещё сложнее, ведь возраст изменился, а коэффициент -нет. В случае с сохранением значений в константы мы снова меняем число в одном месте.

### Задания к семинару

— Написать как можно больше вариантов функции инвертирования массива единиц и нулей за 15 минут (без ветвлений любого рода);
— Сравнить без условий две даты, представленные в виде трёх чисел гггг-мм-дд;

## 4. Специализация: ООП

Инкапсуляция: Классы и объекты (внутренние классы, вложенные классы, static, private/publi final, интерфейс взаимодействия с объектом), перечисления (создание, конструкторы перечислений, объекты перечислений, дополнительные свойства); Наследование: extends, Object (глобальное наследование), protected, преобразование типов, final; Полиморфизм: override, abstract, final;

## 5. Специализация: Тонкости работы

Файловая система и представление данных; Пакеты `java.io`, `java.nio`, `String`, `StringBuilde` string pool, ?JSON/XML?

*Приложения