Техническая специализация Java

(1. Java Core)

Иван Игоревич Овчинников

2022-10-11 (16:45)

Содержание

Всего слов:

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.

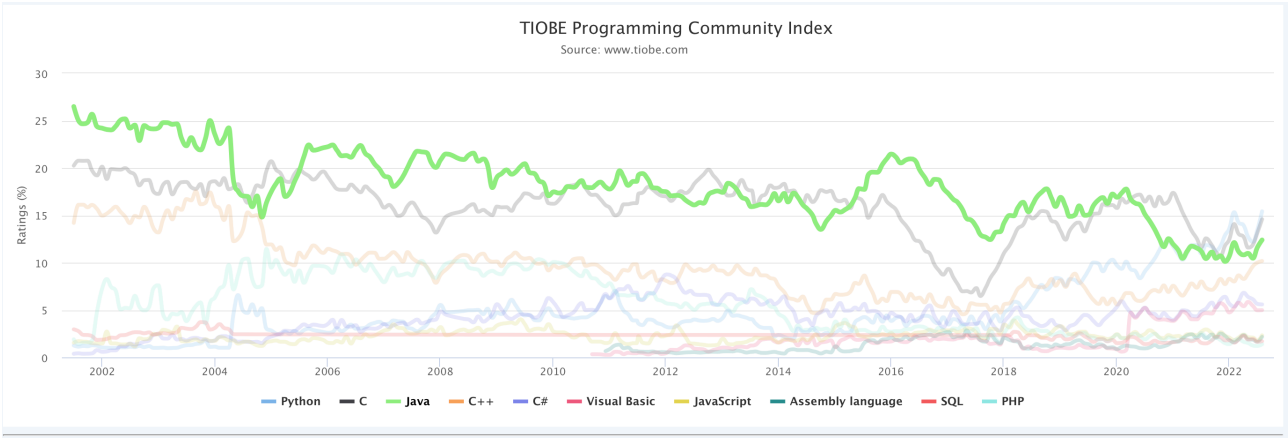


Рис. 1: График популярности языков программирования TIOBE

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

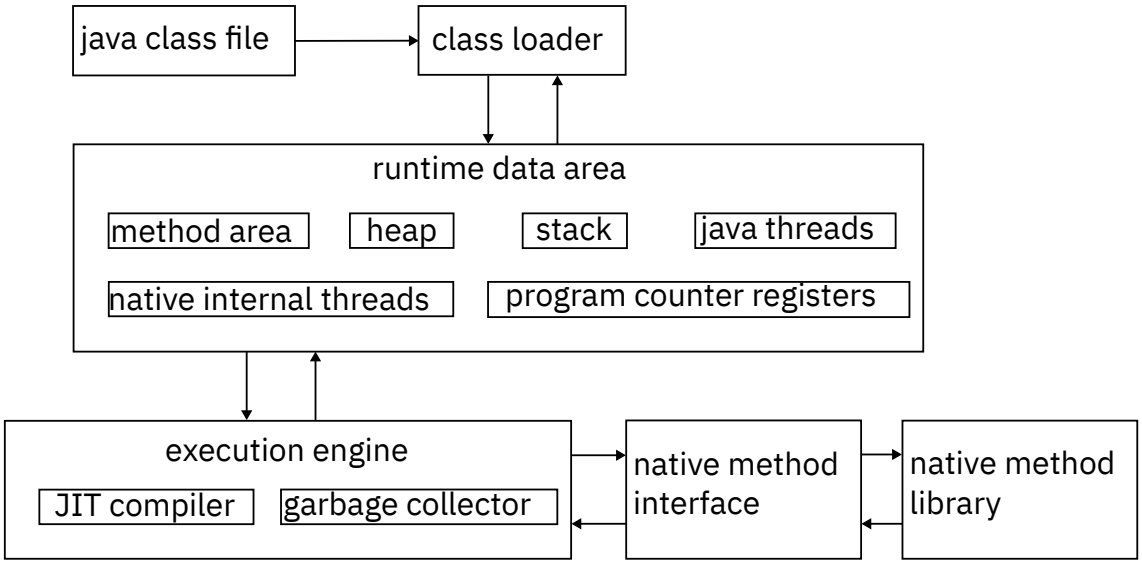


Рис. 2: принцип работы JVM

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 **??** 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.

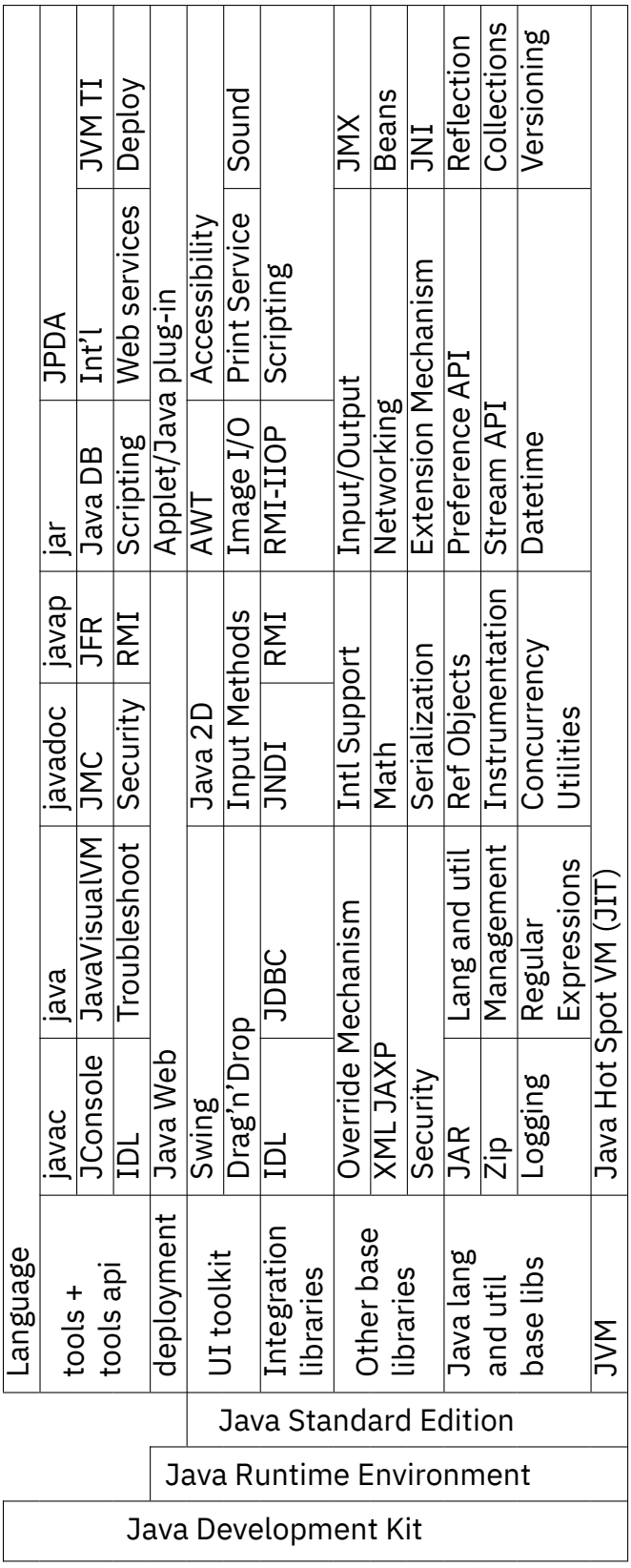


Таблица 1: Общее представление состава JDK

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

```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.

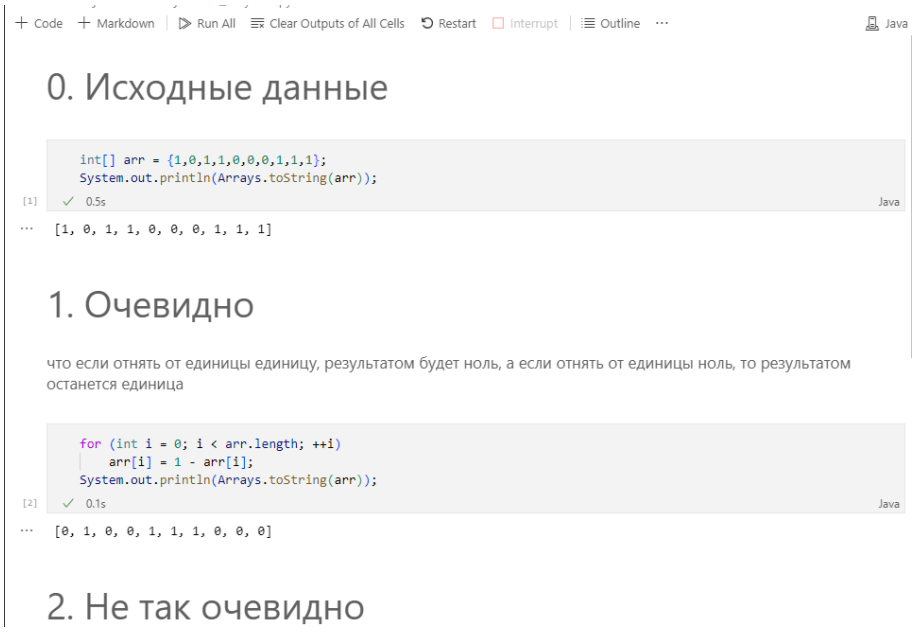


Рис. 3: Пример простого Java проекта в Jupyter Notebook

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. **??**, где папка с выходными (скомпилированными) бинарными файлами пуста, а файл README.md создан для лучшей демонстрации корня проекта.

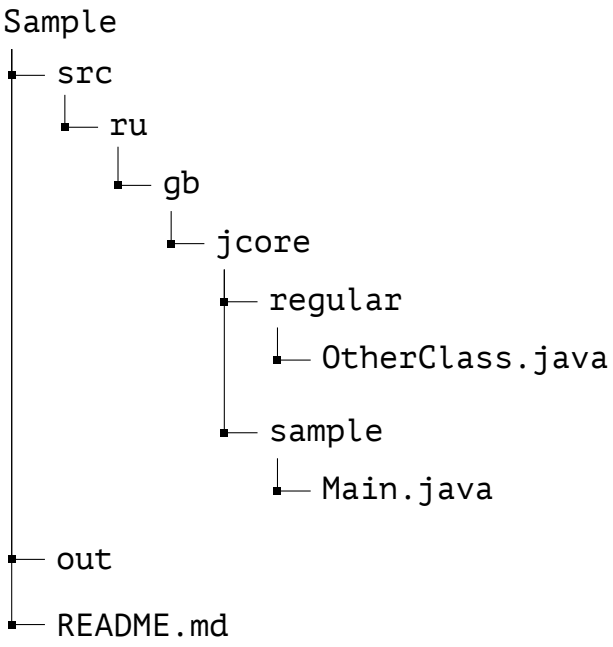


Рис. 4: Структура простого проекта

Содержимое файлов исходного кода представлено ниже.

```
1 package ru.gb.jcore.sample;
2
```

```
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.

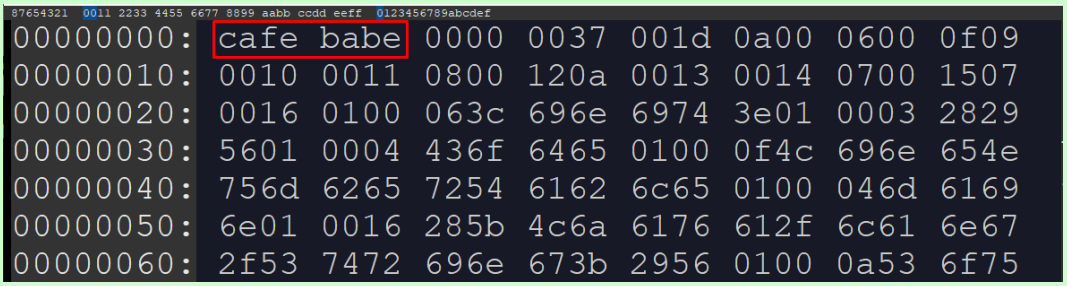
1.7. Отложим мышки в сторону (CLI: сборка, пакеты, запуск)

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».



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.

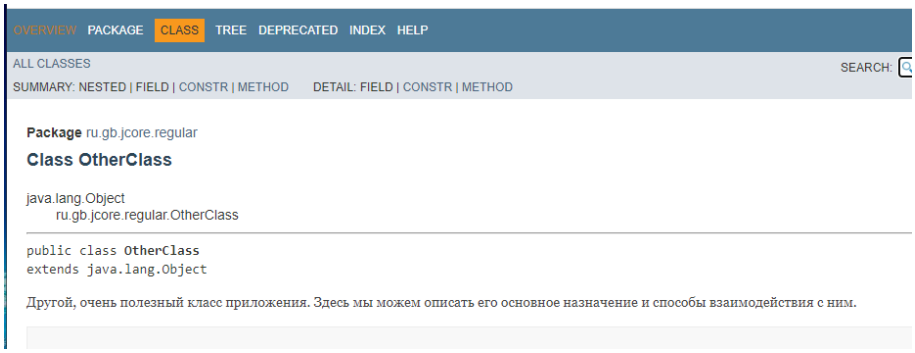


Рис. 5: Часть страницы автосгенерированной документации

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.

Содержание

2. Специализация: данные и функции

2.1. В предыдущем разделе

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

2.2. В этом разделе

Будет рассмотрен базовый функционал языка, то есть основная встроенная функциональность, такая как математические операторы, условия, циклы, бинарные операторы. Далее способы хранения и представления данных в Java, и в конце способы манипуляции данными, то есть функции (в терминах языка называющиеся методами).

— Метод;
— Типизация;
— Переполнение;
— Инициализация;
— Идентификатор;
— Typecasting;
— Массив;

2.3. Данные

2.3.1. Понятие типов

Хранение данных в Java осуществляется привычным для программиста образом: в переменных и константах.

Относительно типизации языки программирования бывают типизированными и нетипизированными (бестиповыми). Нетипизированные языки не представляют большого интереса в современном программировании.

Отсутствие типизации в основном присуще чрезвычайно старым и низкоуровневым языкам программирования, например, Forth и некоторым ассемблерам. Все данные в таких языках считаются цепочками бит произвольной длины и не делятся на типы. Работа с ними часто труднее, 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** (также можно встретить термин «**сильной**») **явной статической** типизацией.

— Статическая - у каждой переменной должен быть тип, и этот тип изменить нельзя. Этому свойству противопоставляется динамическая типизация;
— Явная - при создании переменной ей обязательно необходимо присвоить какой-то тип, явно написав это в коде. В более поздних версиях языка (с 9й) стало возможным инициализировать переменные типа `var`, обозначающий нужный тип тогда, когда его возможно однозначно вывести из значения справа. 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`.

2.3.2. Антипаттерн «магические числа»

Почти во всех примерах, которые используются для обучения, можно увидеть так называемый антипаттерн - плохой стиль для написания кода. Числа, которые находятся справа от оператора присваивания используются в коде без пояснений. Такой антипаттерн называется «магическое число». Магическое, потому что непонятно, что это за число, почему это число именно такое и что будет, если это число изменить.

Так лучше не делать. Заранее нужно сказать, что рекомендуется помещать все числа в коде в именованные константы, которые хранятся в начале файла. Плюсом такого подхода является возможность легко корректировать значения переменных в достаточно больших проектах.

Например, в вашем коде несколько тысяч строк, а какое-то число, скажем, возраст совершеннолетия, число 18, использовалось несколько десятков раз. При использовании приложения в стране, где совершеннолетием считается 21 год вы должны будете перечитывать весь код в поисках магических «18» и исправить их на «21». В этом вопросе будет также важно не запутаться, действительно ли это 18, которые означают совершеннолетие, а не количество карманов в жилетке Анатолия Вассермана[2].

В случае с константой изменить число нужно в одном месте.

2.4. Примитивные типы данных

Все данные в Java делятся на две основные категории: примитивные и ссылочные. Таблица 2 демонстрирует все восемь примитивных типов языка и их размерности. Чтобы отправить на хранение какие-то данные используется оператор присваивания. Присваивание в программировании - это не тоже самое, что математическое равенство, демонстрирующее тождественность, а полноценная операция.

Все присваивания всегда происходят справа налево, то есть сначала вычисляется правая часть, а потом результат вычислений присваивается левой. Исключений нет, именно поэтому в левой части не может быть никаких вычислений.

[2] мы то знаем, что их 26

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

Таблица 2: Основные типы данных в языке Java

Шесть из восьми типов имеет диапазон значений, а значит основное их отличие в объёме занимаемой памяти. У `double` и `float` тоже есть диапазоны, но они заключаются в точности представления дробной части. Диапазоны означают, что если попытаться положить в переменную меньшего типа большее значение, произойдёт «переполнение переменной».

2.4.1. Переполнение целочисленных переменных

Чем именно чревато переполнение переменной легче показать на примере (по ссылке - расследование крушения ракеты из-за переполнения переменной)

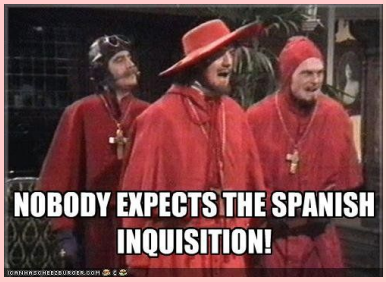
 Переполнение переменных не распознаётся компилятором.

Если создать переменную типа byte, диапазон которого от $[-128, +127]$, и присвоить этой переменной значение 200 произойдёт переполнение, как если попытаться влить пакет молока в напёрсток.

 Переполнение переменной - это ситуация, в которой происходит попытка положить большее значение в переменную меньшего типа.

Важным вопросом при переполнении остаётся следующий: какое в переполненной переменной останется значение? Максимальное, 127? $200 - 127 = 73$? Какой-то мусор? Каждый язык, а зачастую и разные компиляторы одного языка ведут себя в этом вопросе по разному.

В современном мире гигагерцев и терабайтов почти никто не пользуется маленькими типами, но именно из-за этого ошибки переполнения переменных становятся опаснее испанской инквизиции.



2.4.2. Задание для самопроверки

1. Возможно ли объявить в Java целочисленную переменную и присвоить ей дробное значение?
2. Магическое число - это:
   (a) числовая константа без пояснений;
   (b) число, помогающее в вычислениях;
   (c) числовая константа, присваиваемая при объявлении переменной.
3. Переполнение переменной - это:
   (a) слишком длинное название переменной;
   (b) слишком большое значение переменной;
   (c) расширение переменной вследствие записи большого значения.

2.4.3. Бинарное (битовое) представление данных

После разговора о переполнении, нельзя не сказать о том, что именно переполняется. Далее будут представлены сведения которые касаются не только языка Java но и любого другого языка программирования. Эти сведения помогут разобраться в деталях того как хранится значение переменной в программе и как, в целом, происходит работа компьютерной техники.

Все современные компьютеры, так или иначе работают от электричества и являются примитивными по своей сути устройствами, которые понимают только два состояния: есть напряжение в электрической цепи или нет. Эти два состояния принято записывать в виде 1 и 0, соответственно.

Все данные в любой программе - это единицы и нули. Данные в программе на Java не исключение, удобнее всего это явление рассматривать на примере примитивных данных. Поскольку в компьютере можно оперировать только двумя значениями то естественным образом используется двоичная система счисления.

Десятичное	Двоичное	Восьмеричное	Шестнадцатеричное
00	00000	00	0x00
01	00001	01	0x01
02	00010	02	0x02
03	00011	03	0x03
04	00100	04	0x04
05	00101	05	0x05
06	00110	06	0x06
07	00111	07	0x07
08	01000	10	0x08
09	01001	11	0x09
10	01010	12	0x0a
11	01011	13	0x0b
12	01100	14	0x0c
13	01101	15	0x0d
14	01110	16	0x0e
15	01111	17	0x0f
16	10000	20	0x10

Таблица 3: Представления чисел

Двоичная система счисления это система счисления с основанием два. Существуют и другие системы счисления, например, восьмеричная, но сейчас она отходит на второй план полностью уступая своё место шестнадцатеричной системе счисления. Каждая цифра в десятичной записи числа называется разрядом, аналогично в двоичной записи чисел каждая цифра тоже называется разрядом, но для компьютерной техники этот разряд называется битом.

 Одна единица или ноль - это один **бит** передаваемой или хранимой информации.

Биты принято собирать в группы по восемь штук, по восемь разрядов, эти группы называются **байт**. В языке Java возможно оперировать минимальной единицей информации, такой как байт для этого есть соответствующий тип. Диапазон байта, согласно таблицы $[-128, +127]$, то есть байт информации может в себе содержать ровно 256 значений. Само число 127 в двоичной записи это семиразрядное число, все разряды которого единицы (то есть байт выглядит как `01111111`). Последний, восьмой, самый старший бит, определяет знак числа[3]. Достаточно знать формулу расчёта записи отрицательных значений:

1. в прямой записи поменять все нули на единицы и единицы на нули;
2. поставить старший бит в единицу.

Так возможно получить на единицу меньшее отрицательное число, то есть преобразовав 0 получим -1, 1 будет -2, 2 станет -3 и так далее.

Числа бóльших разрядностей могут хранить бóльшие значения, теперь преобразование диапазонов из десятичной системы счисления в двоичную покажет что `byte` это один байт,

[3] Здесь можно начать долгий и скучный разговор о схемотехнике и хранении отрицательных чисел с применением техники дополнительного кода.

`short` это два байта, то есть 16 бит, `int` это 4 байта то есть 32 бита, а `long` это 8 байт или 64 бита хранения информации.

2.4.4. Задания для самопроверки

1. Возможно ли число 3000000000 (3 миллиарда) записать в двоичном представлении?
2. Как вы думаете, почему шестнадцатеричная система счисления вытеснила восьмеричную?

2.4.5. Целочисленные типы

Целочисленных типов четыре, и они занимают 1, 2, 4 и 8 байт.

> Технически, целочисленых типов пять, но `char` устроен чуть сложнее других, поэтому не рассматривается в этом разделе.

Значения в целочисленных типах могут быть только целые, никак и никогда невозможно присвоить им дробных значений. Про эти типы следует помнить следующее:

— `int` - это самый часто используемый тип. Если сомневаетесь, какой целочисленный тип использовать, используйте `int`;
— все целые числа, которые пишутся в коде - это `int`, даже если вы пытаетесь их присвоить переменной другого типа.

Как `int` преобразуется в меньше типы? Если написать цифрами справа число, которое может поместиться в переменную меньшего типа слева, то статический анализатор кода его пропустит, а компилятор преобразует в меньший тип автоматически (строка 9 на рис. 6).

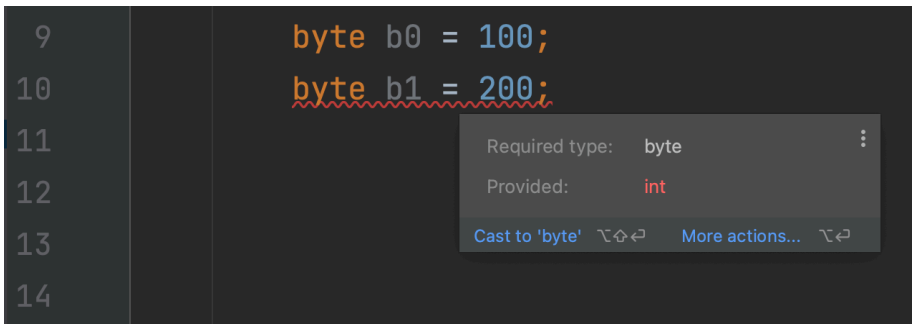


Рис. 6: Присваивание валидных и переполняющих значений

Как видно, к маленькому `byte` успешно присваивается `int`. Если же написать число которое больше типа слева и, соответственно, поместиться не может, среда разработки выдает предупреждение компилятора, что ожидался `byte`, а передан `int` (строка 10 рис 6).

Часто нужно записать в виде числа какое-то значение большее чем может принимать `int`, и явно присвоить начальное значение переменной типа `long`.

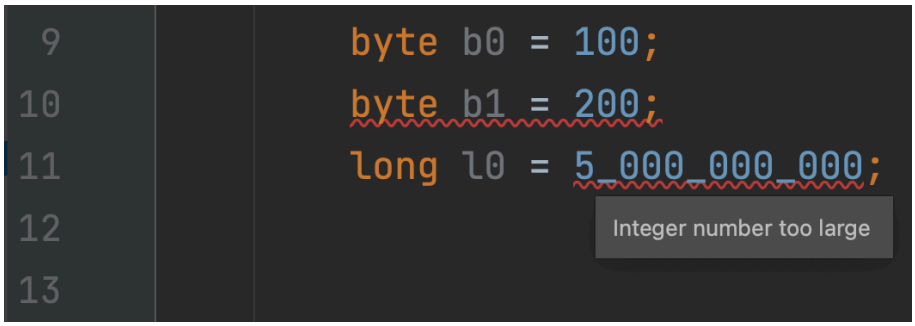


Рис. 7: Попытка инициализации переменной типа `long`

В примере на рис. 7 показана попытка присвоить значение 5000000000 переменной типа `long`. Из текста ошибки ясно, что невозможно положить такое большое значение в переменную типа `int`, а это значит, что справа `int`. Почему большой `int` без проблем присваивается к маленькому байту?

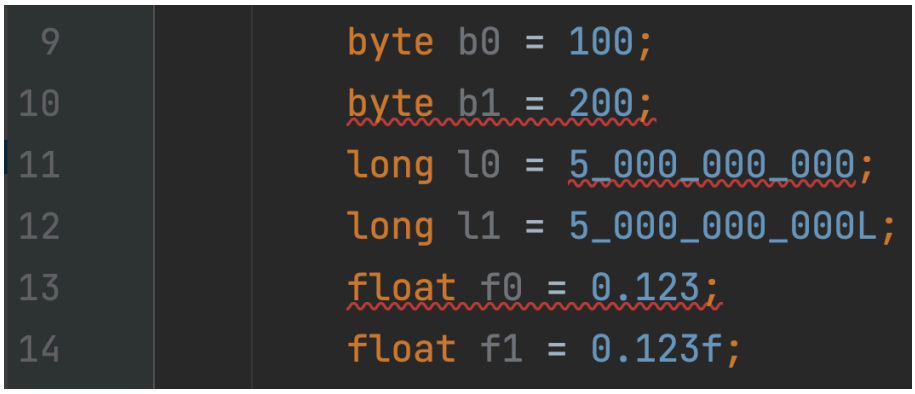


Рис. 8: Решение проблемы переполнения числовых констант

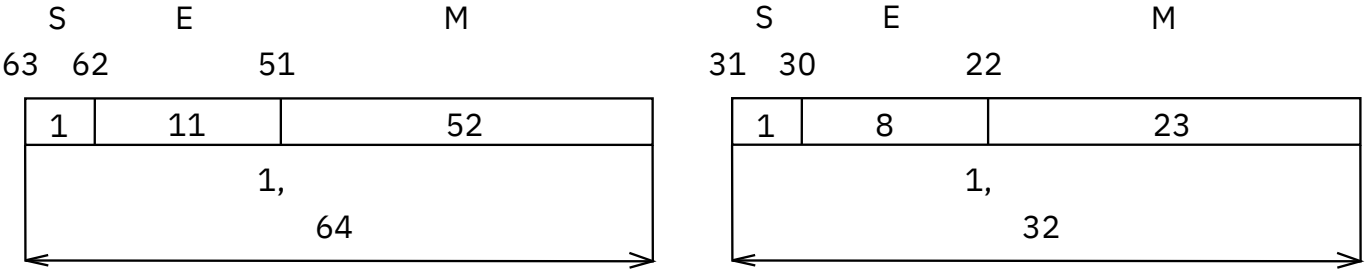
На рис. 8 продемонстрировано, что аналогичная ситуация возникает с типами `float` и `double`. Все дробные числа, написанные в коде - это `double`, поэтому положить их во `float` без дополнительных усилий невозможно. В этих случаях к написанному справа числу нужно добавить явное указание на его тип. Для `long` пишем `L`, а для `float` - `f`. Чаще всего `L` пишут заглавную, чтобы подчеркнуть, что тип больше, а `f` пишут маленькую, чтобы подчеркнуть, что мы уменьшаем тип. Но регистр в этом конкретном случае значения не имеет, можно писать и так и так.

2.4.6. Числа с плавающей запятой (точкой)

Как видно из таблицы 2, два из восьми типов не имеют диапазонов значений. Это связано с тем, что диапазоны значений флоута и дабла заключаются не в величине возможных хранимых чисел, а в точности этих чисел после запятой.

Числа с плавающей запятой в англоязычной литературе называются числа с плавающей точкой (от англ. floating point). Такое различие связано с тем, что в русскоязычной литературе принято отделять дробную часть числа запятой, а в европейской и американской - точкой.

Хранение чисел с плавающей запятой[4] работает по стандарту IEEE 754 (1985 г). Для работы с числами с плавающей запятой на аппаратурном уровне к обычному процессору добавляют математический сопроцессор (FPU, floating point unit).



(a) double (b) float

Рис. 9: Типы с плавающей запятой

Рисунок 9 демонстрирует, как распределяются биты в числах с плавающей запятой разных разрядностей, где S - Sign (знак), E - Exponent (8(11) разрядов поля порядка, экспонента), M - Mantissa (23(52) бита мантиссы, дробная часть числа).

Если попытаться уложить весь стандарт в два предложения, то получится примерно следующее: получить число в соответствующих разрядностях возможно по формулам:

$$F_{32} = (-1)^S \times 2^{E-127} \times (1 + \frac{M}{2^{23}})$$
$$F_{64} = (-1)^S \times 2^{E-1023} \times (1 + \frac{M}{2^{52}})$$

Например: $+0,5 = 2^{-1}$ поэтому, число будет записано как

`0_01111110_00000000000000000000000`, то есть знак $= 0$, мантисса $= 0$, порядок = $127 - 1 = 126$, чтобы получить следующие результаты вычислений:

-1^0 положительный знак, умножить на порядок
$2^{126-127=-1} = 0,5$ и умножить на мантиссу
$1 + 0$. То есть, $-1^0 \times 2^{-1} \times (1 + 0) = 0,5$.
Отсюда становится очевидно, что чем сложнее мантисса и чем меньше порядок, тем более точные и интересные числа мы можем получить.

Возьмём для примера число $-0,15625$, чтобы понять как его записывать, откинем знак, это будет единица в разряде, отвечающем за знак, и посчитаем мантиссу с порядком. Представим число как положительное и будем от него последовательно отнимать числа, являющиеся отрицательными степенями двойки, чтобы получить максимально близкое к нулю значение.

[4] хорошо и подробно, но на C в посте на Хабре.

$$2^1 = 2$$
$$2^0 = 1.0$$
$$2^{-1} = 0.5$$
$$2^{-2} = 0.25$$
$$2^{-3} = 0.125$$
$$2^{-4} = 0.0625$$
$$2^{-5} = 0.03125$$
$$2^{-6} = 0.015625$$
$$2^{-7} = 0.0078125$$
$$2^{-8} = 0.00390625$$

Очевидно, что -1 и -2 степени отнять не получится, поскольку мы явно уходим за границу нуля, а вот -3 прекрасно отнимается, значит порядок будет $127 - 3 = 124$, осталось понять, что получится в мантиссе.

Видим, что оставшееся после первого вычитания $(0,15625 - 0,125)$ число - это 2^{-5}. Значит в мантиссе пишем `01` и остальные нули, то есть слева направо указываем, какие степени после -3 будут нужны. -4 не нужна, а -5 нужна.

Получится, что

$$(-1)^1 \times 2^{(124-127)} \times (1 + \frac{2097152}{2^{23}}) = 1,15652$$

или, тождественно,

$$(-1)^1 \times 1,01e-3 =$$
$$1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} =$$
$$1 \times 0,125 + 0 \times 0,0625 + 1 \times 0,03125 =$$
$$0,125 + 0,03125 = 0,15625$$

Так число с плавающей запятой возможно посчитать двумя способами: по приведённой формуле, или последовательно складывая разряды мантиссы умноженные на двойку в степени порядка, уменьшая порядок на каждом шагу.

К особенностям работы чисел с плавающей запятой можно отнести:

— возможен как положительный, так и отрицательный ноль (в целых числах ноль всегда положительный);
— есть огромная зона, отмеченная на рисунке 10, которая являет собой непредставимые числа, слишком большие для хранения внутри такой переменной или настолько маленькие, что мнимая единица в мантиссе отсутствует;
— в таком числе можно хранить значения положительной и отрицательной бесконечности;
— при работе с такими числами появляется понятие не-числа, при этом важно помнить, что `NaN != NaN`.

2.4.7. Задания для самопроверки

1. Сколько байт данных занимает самый большой целочисленный тип?

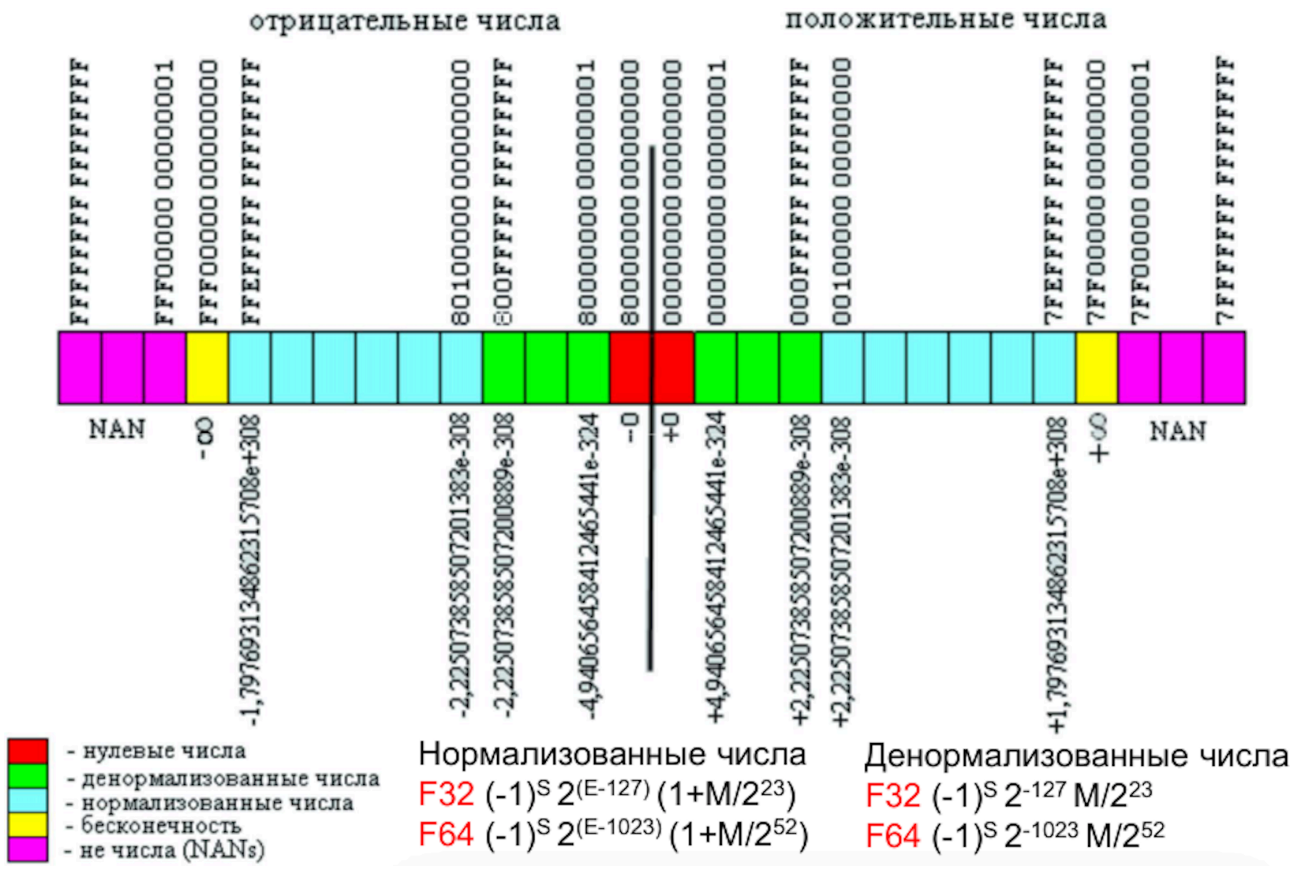


Рис. 10: Особенности работы с числами с плавающей запятой

2. Почему нельзя напрямую сравнивать целочисленные данные и числа с плавающей запятой, даже если там точно лежит число без дробной части?
3. Внутри переполненной переменной остаётся значение:
   (a) переданное - максимальное для типа;
   (b) максимальное для типа;
   (c) не определено.

2.4.8. Символы и булевы

Шесть из восьми примитивных типов могут иметь как положительные, так и отрицательные значения, они называются **«знаковые»** типы. В таблице есть два типа, у которых есть диапазон но нет отрицательных значений, это `boolean` и `char`

Булев тип хранит значение `true` или `false`. На собеседованиях иногда спрашивают, сколько места занимает `boolean`. В Java объём хранения не определён и зависит от конкретной JVM, обычно считают, что это один байт.

Тип `char` единственный беззнаковый целочисленный тип в языке, то есть его старший разряд хранит полезное значение, а не признак положительности. Тип целочисленный но по умолчанию среда исполнения интерпретирует его как символ по таблице utf-8 (см фрагмент в таблице 4). В языке Java есть разница между одинарными и двойными кавычками. В одинарных кавычках всегда записывается символ, который на самом деле является целочисленным значением, а в двойных кавычках всегда записывается строка, которая фактически является экземпляром класса `String`. Поскольку типизация строгая, то невозможно

dec	hex	val	dec	hex	val	dec	hex	val	dec	hex	val
000	0x00	(nul)	032	0x20	□	064	0x40	@	096	0x60	‘
001	0x01	(soh)	033	0x21	!	065	0x41	A	097	0x61	a
002	0x02	(stx)	034	0x22	”	066	0x42	B	098	0x62	b
003	0x03	(etx)	035	0x23	#	067	0x43	C	099	0x63	c
004	0x04	(eot)	036	0x24	$	068	0x44	D	100	0x64	d
005	0x05	(enq)	037	0x25	%	069	0x45	E	101	0x65	e
006	0x06	(ack)	038	0x26	&	070	0x46	F	102	0x66	f
007	0x07	(bel)	039	0x27	'	071	0x47	G	103	0x67	g
008	0x08	(bs)	040	0x28	(	072	0x48	H	104	0x68	h
009	0x09	(tab)	041	0x29	)	073	0x49	I	105	0x69	i
010	0x0A	(lf)	042	0x2A	*	074	0x4A	J	106	0x6A	j
011	0x0B	(vt)	043	0x2B	+	075	0x4B	K	107	0x6B	k
012	0x0C	(np)	044	0x2C	’	076	0x4C	L	108	0x6C	l
013	0x0D	(cr)	045	0x2D	-	077	0x4D	M	109	0x6D	m
014	0x0E	(so)	046	0x2E	.	078	0x4E	N	110	0x6E	n
015	0x0F	(si)	047	0x2F	/	079	0x4F	O	111	0x6F	o
016	0x10	(dle)	048	0x30	0	080	0x50	P	112	0x70	p
017	0x11	(dc1)	049	0x31	1	081	0x51	Q	113	0x71	q
018	0x12	(dc2)	050	0x32	2	082	0x52	R	114	0x72	r
019	0x13	(dc3)	051	0x33	3	083	0x53	S	115	0x73	s
020	0x14	(dc4)	052	0x34	4	084	0x54	T	116	0x74	t
021	0x15	(nak)	053	0x35	5	085	0x55	U	117	0x75	u
022	0x16	(syn)	054	0x36	6	086	0x56	V	118	0x76	v
023	0x17	(etb)	055	0x37	7	087	0x57	W	119	0x77	w
024	0x18	(can)	056	0x38	8	088	0x58	X	120	0x78	x
025	0x19	(em)	057	0x39	9	089	0x59	Y	121	0x79	y
026	0x1A	(eof)	058	0x3A	:	090	0x5A	Z	122	0x7A	z
027	0x1B	(esc)	059	0x3B	;	091	0x5B	[	123	0x7B	{
028	0x1C	(fs)	060	0x3C	<	092	0x5C	\	124	0x7C	|
029	0x1D	(gs)	061	0x3D	=	093	0x5D	]	125	0x7D	}
030	0x1E	(rs)	062	0x3E	>	094	0x5E	^	126	0x7E	~
031	0x1F	(us)	063	0x3F	?	095	0x5F	_	127	0x7F	\DEL

Таблица 4: Фрагмент UTF-8 (ASCII) таблицы

записать в `char` строки, а в строки числа.

В Java есть три основных понятия, связанных с данными переменными и использованием значений: объявление, присваивание, инициализация.

Для того чтобы *объявить* переменную, нужно написать её тип и название, также часто вместо названия можно встретить термин идентификатор.
Далее в любой момент можно *присвоить* этой переменной значение, то есть необходимо написать идентификатор использовать оператор присваивания и справа написать значение, которое вы хотите присвоить данной переменной, поставить в конце строки точку с запятой.
Также существует понятие *инициализации* - это когда объединяются на одной строке объявление и присваивание.

2.4.9. Преобразование типов

Java - это язык со строгой статической типизацией, но преобразование типов в ней всё равно есть. Простыми словами, преобразование типов - это когда компилятор видит, что типы переменных по разные стороны присваивания разные, начинает разрешать это противоречие. Преобразование типов бывает явное и неявное.

В разговоре или в сообществах можно услышать или прочитать термины тайпкастинг, кастинг, каст, кастануть, и другие производные от английского typecasting.

Неявное преобразование типов происходит, когда присваиваются числа переменным меньшей размерности, чем `int`. Число справа это `int`, а значит 32 разряда, а слева, например, `byte`, и в нём всего 8 разрядов, но ни среда ни компилятор не поругались, потому что значение в большом `int` не превысило 8 разрядов маленького `byte`. Итак неявное преобразование типов происходит в случаях, когда, «всё и так понятно». В случае, если неявное преобразование невозможно, статический анализатор кода выдаёт ошибку, что ожидался один тип, а был дан другой.

Явное преобразование типов происходит, когда мы явно пишем в коде, что некоторое значение должно иметь определённый тип. Этот вариант приведения типов тоже был рассмотрен, когда к числам дописывались типовые квалификаторы `L` и `f`. Но чаще всего случается, что происходит присваивание переменным не тех значений, которые были написаны в тексте программы, а те, которые получились в результате каких-то вычислений.

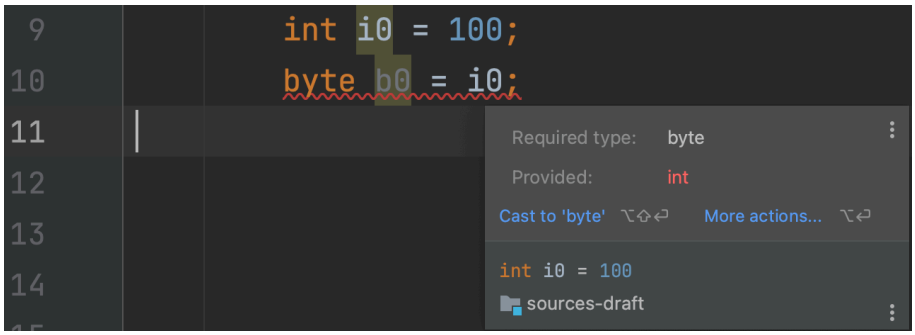


Рис. 11: Ошибка приведения типов

На рис. 12 приведён простейший пример, в котором очевидно, что внутри переменной `i0` содержится значение, не превышающее одного байта хранения, а значит возможно явно сообщить компилятору, что значение точно поместится в `byte`. *Явно преобразовать типы*. Для этого нужно в правой части оператора присваивания перед идентификатором переменной в скобках добавить название типа, к которому необходимо преобразовать значение этой переменной.

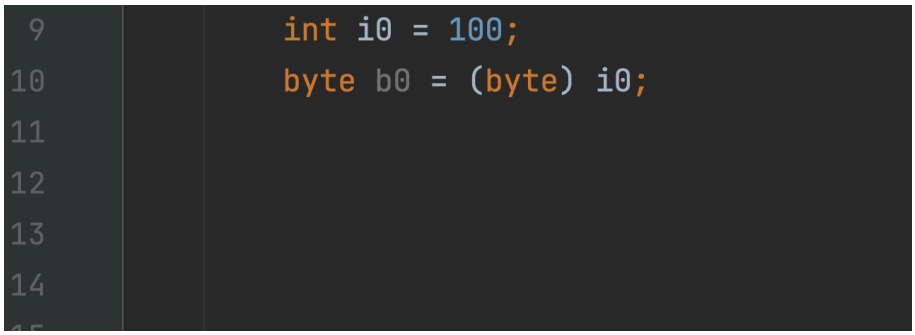


Рис. 12: Верное приведение типов

2.4.10. Константность

Constare - (лат. стоять твёрдо). Константность это свойство неизменяемости. В Java ключевое слово `const` не реализовано, хоть и входит в список ключевых, зарезервированных. Константы создаются при помощи ключевого слова `final`. Ключевое слово файнал возможно применять не только с примитивами, но и со ссылочными типами, методами, классами.

Константа - это переменная или идентификатор с конечным значением.

2.4.11. Задания для самопроверки

1. Какая таблица перекодировки используется для представления символов?
2. Каких действий требует от программиста явное преобразование типов?
3. какое значение будет содержаться в переменной a после выполнения строки int a = 10.0f/3.0f;

2.5. Ссылочные типы данных, массивы

Ссылочные типы данных - это все типы данных, кроме восьми перечисленных примитивных. Самым простым из ссылочных типов является массив. Фактически массив выведен на уровень языка и не имеет специального ключевого слова.

Ссылочные типы отличаются от примитивных местом хранения информации. В примитивах данные хранятся там, где существует переменная и где написан её идентификатор, а по идентификатору ссылочного типа хранится не значение, а ссылка. Ссылку можно представить как ярлык на рабочем столе, то есть очевидно, что непосредственная информация хранится не там, где написан идентификатор. Такое явное разделение идентификатора переменной и данных важно помнить и понимать при работе с ООП.

Массив - это единая, сплошная область данных, в связи с чем в массивах возможно осуществление доступа по индексу

Самый младший индекс любого массива - ноль, поскольку **индекс** - это значение смещения по элементам относительно начального адреса массива. То есть, для получения самого первого элемента нужно сместиться на ноль шагов. Очевидно, что самый последний элемент в массиве из десяти значений, будет храниться по девятому индексу.

Массивы возможно создавать несколькими способами (листинг 1). В общем виде объявление - это тип, квадратные скобки как обозначение того, что это будет массив из переменных этого типа, идентификатор (строка 1). Инициализировать массив можно либо ссылкой на другой массив (строка 2), пустым массивом (строка 3) или заранее заданными значениями, записанными через запятую в фигурных скобках (строка 4). Присвоить в процессе работы идентификатору возможно только значение ссылки из другого идентификатора или новый пустой массив.

Листинг 1: Объявление массива

```
1 int[] array0;
2 int[] array1 = array0;
3 int[] array2 = new int[5];
4 int[] array3 = {5, 4, 3, 2, 1};
5
6 array2 = {1, 2, 3, 4, 5}; //
```

Никак и никогда нельзя присвоить идентификатору целый готовый массив в процессе работы, нельзя стандартными средствами переприсвоить ряд значений части массива (так называемые слайсы или срезы).

Массивы бывают как одномерные, так и многомерные. Многомерный массив - это всегда массив из массивов меньшего размера: двумерный массив - это массив одномерных, трёхмерный - массив двумерных и так далее. Правила инициализации у них не отличаются. Преобразовать тип массива нельзя никогда, но можно преобразовать тип каждого отдельного элемента при чтении. Это связано с тем, что под массивы сразу выделяется непрерывная область памяти, а со сменой типа всех значений массива эту область нужно будет или значительно расширять или значительно сужать.

Ключевое слово `final` работает только с идентификатором массива, то есть не запрещает изменять значения его элементов.

Если логика программы предполагает создание нижних измерений массива в процессе работы программы, то при инициализации массива верхнего уровня не следует указывать размерности нижних уровней. Это связано с тем, что при инициализации, Java сразу выделяет память под все измерения, а присваивание нижним измерениям новых ссылок на создаваемые в процессе работы массивы, будет пересоздавать области памяти, получается небольшая утечка памяти.

Прочитать из массива значение возможно обратившись к ячейке массива по индексу. Записать в массив значение возможно обратившись к ячейке массива по индексу, и применив оператор присваивания.

```
int i = array[0];
array[1] = 10;
```

2.5.1. Задания для самопроверки

1. Почему индексация массива начинается с нуля?
2. Какой индекс будет последним в массиве из 100 элементов?
3. Сколько будет создано одномерных массивов при инициализации массива 3x3?

2.6. Базовый функционал языка

2.6.1. Математические операторы

Математические операторы работают как и предполагается - складывают, вычитают, делят, умножают, делают это по приоритетам известным нам с пятого класса, а если приоритет одинаков - слева направо. Специального оператора возведения в степень как в пайтоне нет. Единственное, что следует помнить, что оператор присваивания продолжает быть оператором присваивания, а не является математическим равенством, а значит сначала посчитается всё, что слева, а потом результат попробует присвоиться переменной справа. Припоминаем что там за дела с целочисленным делением и отбрасыванием дробной части.

2.6.2. Условия

Условия представлены в языке привычными `if`, `else if`, `else`, «если», «иначе если», «в противном случае», которые являются единым оператором выбора, то есть если исполнение программы пошло по одной из веток, то в другую ветку условия программа точно не зайдёт. Каждая ветвь условного оператора - это отдельный кодовый блок со своим окружением и локальными переменными.

Существует альтернатива оператору `else if` - использование оператора `switch`, который позволяет осуществлять множественный выбор между числовыми значениями. У оператора есть ряд особенностей:

— это оператор, состоящий из одного кодового блока, то есть сегменты кода находятся в одной области видимости. Если не использовать оператор `break`, есть риск «проваливаться» в следующие кейсы;
— нельзя создать диапазон значений;
— достаточно сложно создавать локальные переменные с одинаковым названием для каждого кейса.

2.6.3. Циклы

Циклы представлены основными конструкциями:
— `while () {}`
— `do {} while();`
— `for (;;) {}`

Цикл - это набор повторяющихся до наступления условия действий. `while` - самый простой, чаще всего используется, когда нужно описать бесконечный цикл. `do-while` единственный цикл с постусловием, то есть сначала выполняется тело, а затем принимается решение о необходимости зацикливания, используется для ожидания ответов на запрос и возможного повторения запроса по условию. `for` - классический счётный цикл, его почему-то программисты любят больше всего.

Существует также активно пропагандируемый цикл - `foreach`, работает не совсем очевидным образом, для понимания его работы необходимо ознакомиться с ООП и понятием итератора.

2.6.4. Бинарные арифметические операторы

В современных реалиях мегамощных компьютеров вряд ли кто-то задумывается об оптимизации скорости выполнения программы или экономии занимаемой памяти. Но всё меняется, когда программист впервые принимает сложное решение: запрограммировать микроконтроллер или другой «интернет вещей». Там в вашем распоряжении жалкие пара сотен килобайт памяти, если очень повезёт, в которые нужно не только как-то вложить текст программы и исполняемый бинарный код, но и какие-то промежуточные, пользовательские и другие данные, буферы обмена и обработки. Другая ситуация, в которой нужно начинать «думать о занимаемом пространстве» это разработка протоколов передачи данных, чтобы протокол был быстрый, не передавал по сети большие объёмы данных и быстро преобразовывался. На помощь приходит натуральная для информатики система счисления, двоичная.

Манипуляции двоичными данными представлены в Джава следующими операторами:
— & битовое и;
— | битовое или;
— ~ битовое не;
— ^ исключающее или;
— << сдвиг влево;
— >> сдвиг вправо.

Литеральные «и», «или», «не» уже знакомы по условным операторам. Литеральные операции применяются ко всему числовому литералу целиком, а не к каждому отдельному би-

ту. Их особенность заключается в том, как язык программирования интерпретирует числа.

В Java в литеральных операциях может участвовать только тип `boolean`, а C++ воспринимает любой ненулевой целочисленный литерал как истину, а нулевой, соответственно, как ложь.

Логика формирования значения при этом остаётся такой же, как и при битовых операциях.

A	B	AND
0	0	0
0	1	0
1	0	0
1	1	1

A	B	OR
0	0	0
0	1	1
1	0	1
1	1	1

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

A	NOT
0	1
1	0

Таблица 5: Таблицы истинности битовых операторов

Когда говорят о битовых операциях волей-неволей появляется необходимость поговорить о таблицах истинности. В таблице 5 вы видите таблицы истинности для арифметических битовых операций. Битовые операции отличаются тем, что для неподготовленного взгляда они производят почти магические действия, потому что манипулируют двоичным представлением числа.

```
00000100  4        00000100  4        00000100  4        ~00000100  4
&00000111 7        |00000111 7        ^00000111 7        11111011   -5
00000100  4        00000111  7        00000011  3
```

Рис. 13: Бинарная арифметика

Число	Бинарное	Сдвиг
2	000000010	`2 << 0`
4	000000100	`2 << 1`
8	000001000	`2 << 2`
16	000010000	`2 << 3`
32	000100000	`2 << 4`
64	001000000	`2 << 5`
128	010000000	`2 << 6`

Число	Бинарное	Сдвиг
128	010000000	`128 >> 0`
64	001000000	`128 >> 1`
32	000100000	`128 >> 2`
16	000010000	`128 >> 3`
8	000001000	`128 >> 4`
4	000000100	`128 >> 5`
2	000000010	`128 >> 6`

Таблица 6: Битовые сдвиги

С битовыми сдвигами работать гораздо интереснее и выгоднее. Они производят арифметический сдвиг значения слева на количество разрядов, указанное справа, в таблице 6 представлены числа, в битовом представлении это одна единственная единица, находящаяся в разных разрядах числа. Это демонстрация сдвига на один разряд влево, и, как следствие, умножение на два. Обратная ситуация со сдвигом вправо, он является целочисленным делением.

— `X && Y` - литеральная;
— `X || Y` - литеральная;
— `!X` - литеральная;
— `N << K` - $N * 2^K$;
— `N >> K` - $N/2^K$;
— `x & y` - битовая. 1 если оба x = 1 и y = 1;
— `x | y` - битовая. 1 если хотя бы один из x = 1 или y = 1;
— `~x` - битовая. 1 если x = 0;
— `x ^ y` - битовая. 1 если x отличается от y.

2.6.5. Задания для самопроверки

1. Почему нежелательно использовать оператор `switch` если нужно проверить диапазон значений?
2. Возможно ли записать бесконечный цикл с помощью оператора `for`?
3. `2 + 2 * 2 == 2 << 2 >> 1`?

2.7. Функции

Функция - это исполняемый блок кода. Функция, принадлежащая классу называется **методом**.

```
1  void method(int param1, int param2) {
2     //function body
3  }
4
5  public static void main (String[] args) {
6     method(arg1, arg2);
7  }
```

При объявлении функции в круглых скобках указываются параметры, а при вызове - аргументы.

У функций есть правила именования: функция - это переходный глагол совершенного вида в настоящем времени (вернуть, посчитать, установить, создать), часто снабжаемый дополнением, субъектом действия. Методы в Java пишутся lowerCamelCase. Важно, в каком порядке записаны параметры метода, от этого будет зависеть порядок передачи в неё аргументов. Методы обособлены и их параметры локальны, то есть не видны другим функциям.

Нельзя писать функции внутри других функций.

Все аргументы передаются копированием, не важно, копирование это числовой константы, числового значения переменной или хранимой в переменной ссылке на массив. Сам объект в метод не копируется, а копируется только его ссылка.

Возвращаемые из методов значения появляются в том месте, где метод был вызван. Если будет вызвано несколько методов, то весь контекст исполнения первого метода сохраняется, кладётся (на стек) в стопку уже вызванных методов и процессор идёт выполнять только что вызванный второй метод. По завершении вызванного второго метода, мы снимаем со стека лежащий там контекст первого метода, кладём в него вернувшееся из второго метода значение, если оно есть, и продолжаем исполнять первый метод.

Вызов метода - это, по смыслу, тоже самое, что подставить в код сразу его возвращаемое значение.

Сигнатура метода — это имя метода и его параметры. В сигнатуру метода не входит возвращаемое значение. Нельзя написать два метода с одинаковой сигнатурой.

Перегрузка методов - это механизм языка, позволяющий написать методы с одинаковыми названиями и разными оставшимися частями сигнатуры, чтобы получить единообразие при вызове семантически схожих методов с разнотипными данными.

2.8. Практическое задание

1. Написать метод «Шифр Цезаря», с булевым параметром зашифрования и расшифрования и числовым ключом;
2. Написать метод, принимающий на вход массив чисел и параметр n. Метод должен осуществить циклический (последний элемент при сдвиге становится первым) сдвиг всех элементов массива на n позиций;
3. Написать метод, которому можно передать в качестве аргумента массив, состоящий строго из единиц и нулей (целые числа типа `int`). Метод должен заменить единицы в массиве на нули, а нули на единицы и не содержать ветвлений. Написать как можно больше вариантов метода.

Задания к семинару

— Сравнить без условий две даты, представленные в виде трёх чисел гггг-мм-дд;

3. Управление проектом: сборщики проектов

3.1. 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.

3.2. 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;
— DSL
— XML
— Артефакт;
— Репозиторий;
— Прокси;

3.3. Мотивация и схема (зачем это нужно и как это работает)

Реальный проект - это не только непосредственный код, который пишется в IDE[5], но и большое количество всего, что мы при этом используем - библиотеки, фреймворки и т.п. То есть, мы не пишем каждый проект «с чистого листа», использование JRE/JDK нам это явно демонстрирует, поскольку в них есть уже готовые написанные классы, реализующие некоторую функциональность (подробнее на странице 7).

[5]Чаще всего, называется бизнес-кодом или бизнес-логикой

Всё, что так или иначе используется в проекте, кроме непосредственного кода, написанного в IDE, называется **зависимостями проекта**, поскольку код всегда «опирается» на заранее созданный инструментарий. Например, мы просто пишем строку с приветствием миру, не думая, как именно она раскладывается в массив символов. Или мы просто выводим приветствие миру в терминал, а инструментарий берёт на себя открытие вспомогательных потоков и прочие сложности.

Из-за этого, *единый исполняемый файл* проекта[6], включающий в себя все зависимости, может разрастаться до нескольких сотен мегабайт, а о том, чтобы скомпилировать его со всеми зависимостям, и речи быть не может.

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

Они позволяют 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;

[6]Чаще всего, такой файл нужен для быстрого переноса и запуска проекта на устройстве, на котором не установлен JDK

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 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, описывает задачи сборки в виде XML файлов;
— Самый популярный инструмент - это Maven, используется для JavaEE и приложений с использованием Spring Framework, основана на специальном подмножестве XML, называемом POM[7];
— Gradle основной инструмент в мобильной разработке, часто описания сборки на специальном DSL на основе Groovy/Kotlin, оказываются удобнее XML;

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

3.4. 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[8]). В отличие от `make`, утилита Ant имеет только одну зависимость — JRE. Ещё одним важным отличием от `make` является отказ от использования команд операционной системы. Всё что пишется в Ant пишется на XML, что обеспечивает переносимость сценариев между платформами.

Управление процессом сборки происходит посредством XML-сценария, также называемого Build-файлом. В первую очередь этот файл содержит определение проекта, состоящего из отдельных целей (жарг. таргетов, стр. 13)). Цели в терминах Ant сравнимы с процедурами в языках программирования и содержат вызовы команд-заданий (жарг. тасков). Каждое задание представляет собой неделимую, атомарную команду, выполняющую некоторое элементарное действие.

Между целями могут быть определены зависимости[9] — 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

[7] (англ. Project Object Model) модель объектов проекта
[8] от англ task - задача, задание
[9] не путать с зависимостями проекта, здесь речь исключительно о задачах и командах Ant

— `clean` (удаление промежуточных файлов);
— `compile` (компиляция всех классов);
— `deploy` (развёртывание приложения на сервере).

Скачивание и установка, Ant для Linux-подобных ОС выполняется командой установки в пакетном менеджере (для Debian-подобных дистрибутивов `sudo apt install ant`), а для Windows необходимо перейти на сайт https://ant.apache.org, скачать архив с приложением, распаковать его и прописать соответствующий путь в переменные среды. Проверить установку и версию можно командой

```
ant -version
```

Теперь можно написать простой сценарий HelloWorld, для этого необходимо создать файл `build.xml` (см листинг 2). В нём мы указываем следующие данные:

1. версию XML, иначе XML не работает;
2. название проекта в теге `project` параметр `name`;
3. таргет по умолчанию в теге `project` параметр `default`;
4. внутри тега `project` описания таргетов, в теге `target` параметр `name` указывает имя цели;
5. внутри тега `target` команды, которые необходимо выполнить, например, <echo>.

Листинг 2: сценарий Ant echo «Hello, World!»

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

Затем, нужно создать каталог `hello` и сохранить туда только что созданный файл с именем `build.xml`, который содержит сценарий. Далее надо перейти в каталог и вызвать `ant`.

Следует обратить внимание, что команды навигации (а также, копирования, создания, редактирования файлов) в терминале могут (и, скорее всего, будут) отличаться.

Полный перечень команд, например:

```
mkdir hello
cd hello
cp ../build.xml .
ant
```

В результате выполнения получим следующий вывод.

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

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

Система сборки нашла файл сценария с именем по умолчанию (`build.xml`) и выполнила цель указанную по умолчанию (`hello`). 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.

Ниже будет представлен пример build-файла, а в качестве проекта для сборки возьмём из прошлого урока с двумя классами. Файл для него будет выглядеть следующим образом: В случае, если у вас не сохранился проект с первого урока, то создайте структуру как на рис. **??**. После чего, можно приступить к написанию билд-файла, который будет находиться в корне проекта (на одном уровне с out и src). Выглядеть он будет следующим образом:

```
<?xml version="1.0"?>
<project name="Sample" default="run">
    <property name="src.dir" value="src/ru/gb/jcore" />
    <property name="build.dir" value="out" />
    <property name="classes.dir" value="${build.dir}/classes" />
    <property name="jar.dir" value="${build.dir}/jar" />
    <property name="main-class" value="ru.gb.jcore.sample.Main" />

    <path id="classpath">
        <fileset dir="${jar.dir}">
            <include name="*.jar" />
        </fileset>
        <fileset dir="${classes.dir}">
            <include name="/*.class" />
        </fileset>
    </path>

    <target name="clean">
        <delete dir="${classes.dir}" />
    </target>

    <target name="compile">
        <mkdir dir="${classes.dir}" />
        <mkdir dir="${jar.dir}" />
        <javac destdir="${classes.dir}" includeAntRuntime="false">
            <src path="${src.dir}/regular"/>
            <src path="${src.dir}/sample"/>
            <classpath refid="classpath" />
        </javac>
    </target>

    <target name="jar" depends="compile">
        <jar destfile="${jar.dir}/${ant.project.name}.jar" basedir="${classes.dir}">
            <manifest>
                <attribute name="Main-Class" value="${main-class}" />
                <attribute name="Class-Path" value="${jar.dir}"/>
            </manifest>
        </jar>
    </target>

    <target name="run" depends="jar">
        <java classname="${main-class}">
```

```
43              <classpath refid="classpath" />
44          </java>
45      </target>
46
47      <target name="clean-build" depends="clean,jar" />
48      <target name="main" depends="clean,run" />
49  </project>
```

Теперь по порядку, что есть что. Первым делом мы встречаем тег xml, в котором указывается его версия, а затем тег project с указанием имени проекта и команды по умолчанию. После, в build-файле можно заметить тег property, который по своей натуре является переменной, чтобы сократить те или иные пути до папок/файлоы. Первая переменная с идентификатором (именем) src.dir имеет значение, которое ведёт до пакетов с файлами .java. Затем build.dir, которая обозначает папку, куда будут помещаться файлы после компиляции. classes.dir - путь до папок со сгенерированными файлами .class. jar.dir - директория с бинарными файлами. main-class - это местоположение главного класса приложения, чтобы потом указать .jar-файлу точку входа. Наш сценарий содержит шесть target (команд):

1. Clean - удаляет папки с результатами компиляции;
2. Compile - создаёт две директории, одна для компиляции файлов .java, другая для создания .jar файла, а также указывается какой из двух классов куда будет скомпилирован;
3. Jar - она имеет свойство depends, что означает зависимость от указанной команды, а в нашем случае это команда compile, т.е. вначале выполнится compile, после чего команда jar. В подтеге jar присутствует настройка манифеста, там указывается путь до главного класса, в качестве входа в программу, а также местоположение бинарного файла;
4. Run - зависит от команды jar и является командой по умолчанию для проекта;
5. Clean-build - зависит от команд clean и jar. По своей сути, она выполняет просто эти две команды;
6. Main - выполняет команды clean и run.

Также, имеется тег path - в нашем случае он указывает где и какие файлы будут находиться после компиляции. После чего, можно выполнить команду ant run, а вывод будет следующим:

```
Buildfile: D:\Desktop\Java\GB\Sample\build.xml

compile:
    [javac] Compiling 2 source files to D:\Desktop\Java\GB\Sample\out\class

jar:
      [jar] Building jar: D:\Desktop\Java\GB\Sample\out\jar\Sample.jar

run:
     [java] Hello, world!
     [java] Here is your number: 4.
```

```
BUILD SUCCESSFUL
Total time: 0 seconds
```

3.5. Репозитории, артефакты, конфигурации

3.6. Классический подход (Maven)

3.7. Всем давно надоел XML (Gradle)

3.8. Собственные прокси, хостинг и закрытая сеть

3.9. Немного экзотики (Bazel)

Ant, Ant+Ivy

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

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 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, 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>Hello, 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”/> </dep
</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.

4. Специализация: ООП

5. Специализация: Тонкости работы

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

*Приложения