Техническая специализация Java

(1. Java Core)

Иван Игоревич Овчинников

2022-12-15 (10:46)

Содержание

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.

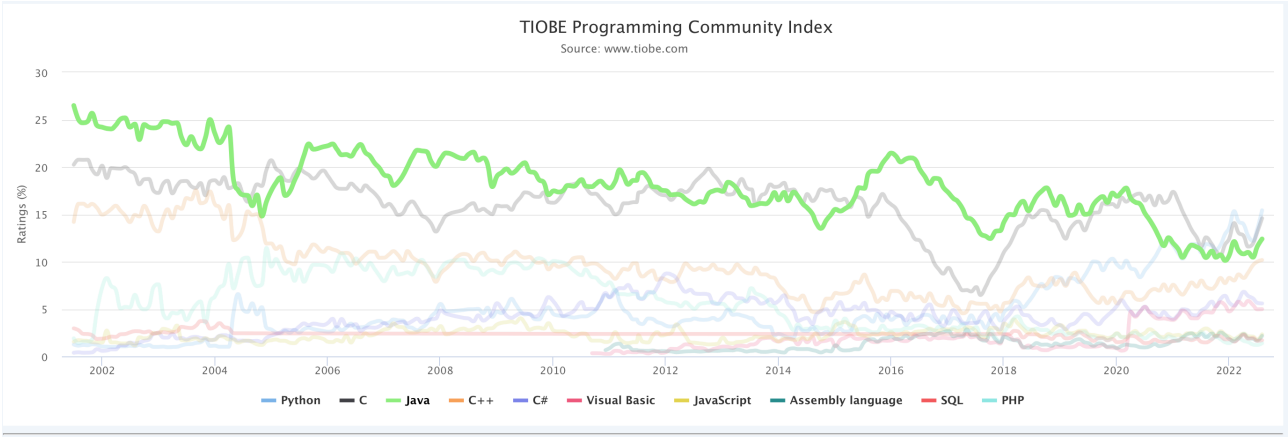


Рис. 1: График популярности языков программирования TIOBE

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 исполь-

зовать 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 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 кэшируется 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

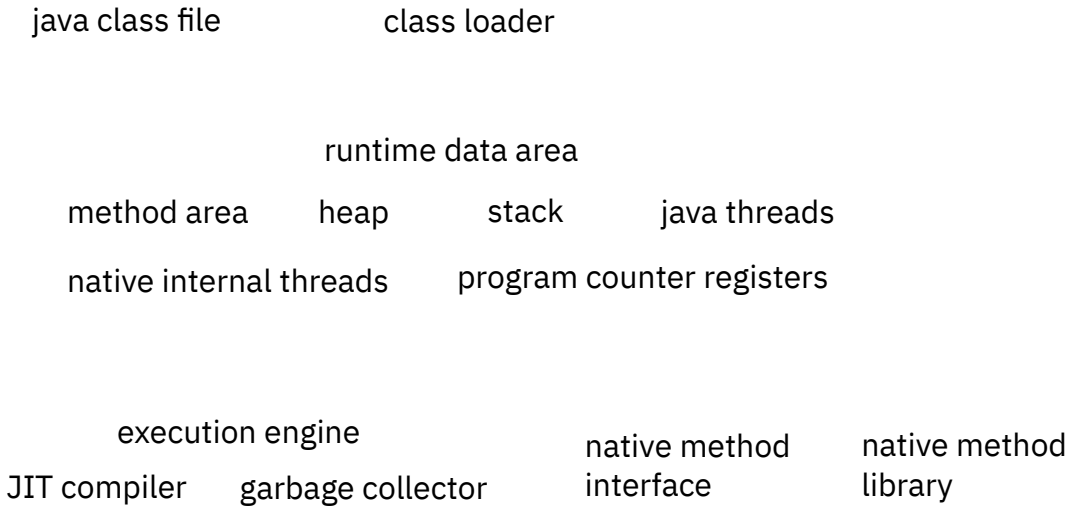


Рис. 2: Принцип работы JVM

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 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.

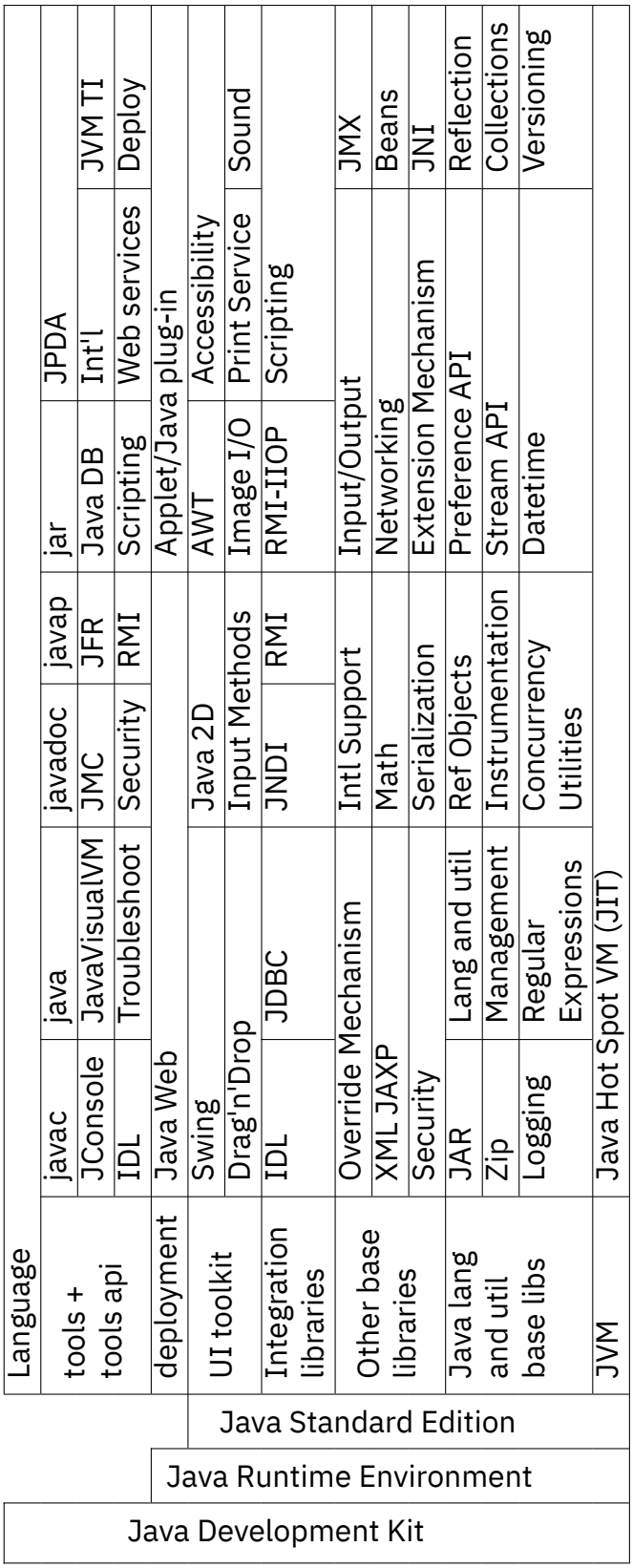


Таблица 1: Общее представление состава JDK

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.

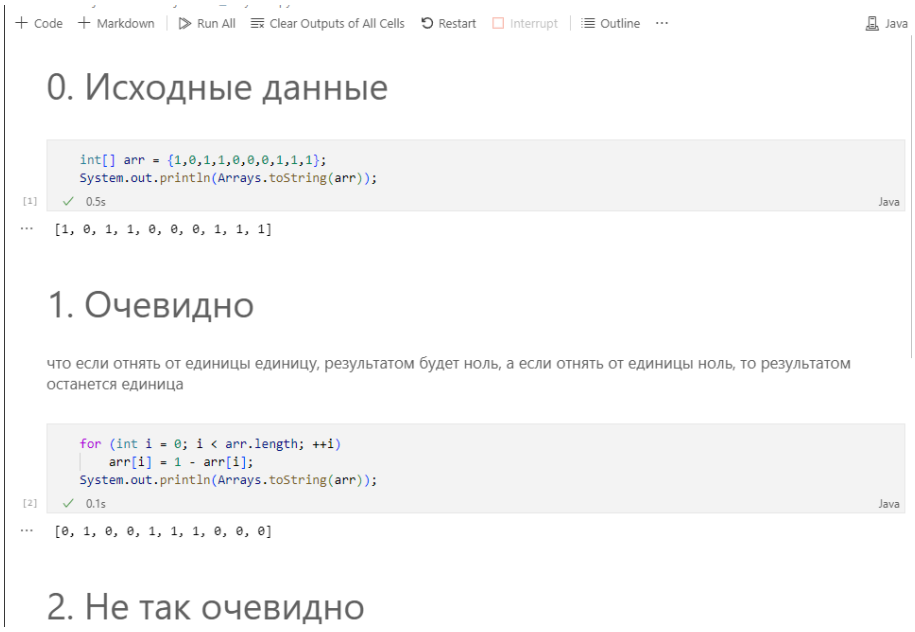


Рис. 3: Пример простого Java проекта в Jupyter Notebook

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

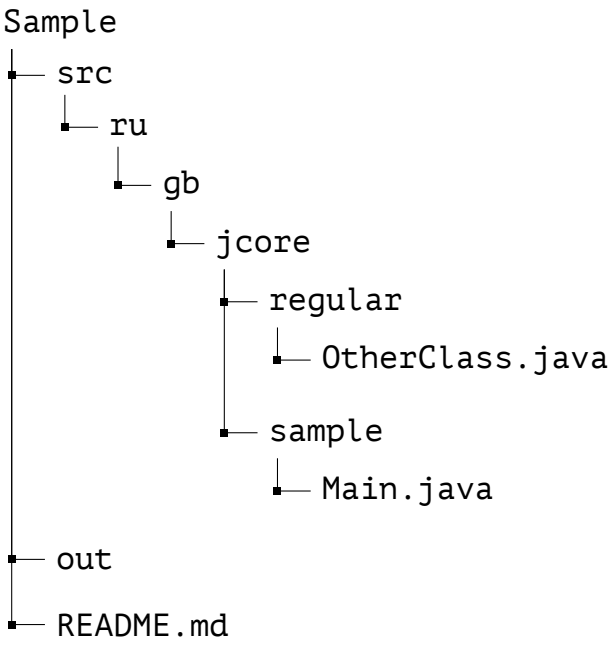


Рис. 4: Структура простого проекта

Содержимое файлов исходного кода представлено ниже.

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

```java
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.

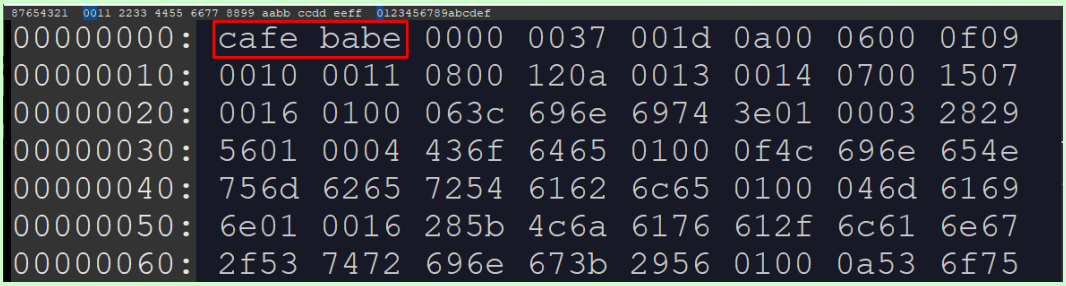
1.7. Отложим мышки в сторону (CLI: сборка, пакеты, запуск)

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 шестнадцатеричном 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.

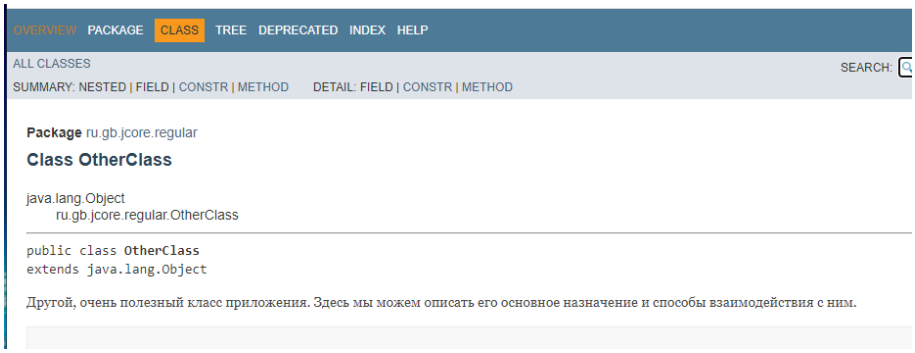


Рис. 5: Часть страницы автосгенерированной документации

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 запускается.

```
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.

Практическое 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 пользователя, 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 девятой) стало возможным инициализировать переменные типа `var`, обозначающего нужный тип тогда, когда его возможно однозначно вывести из значения справа[2]. Бывают языки с неявной типизацией, например, Python.
— Строгая(сильная) -- невозможно смешивать разнотипные данные. С другой стороны, существует, например, 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.

В реальных проектах 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[3].

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] аналог типа `auto` в языке C++
[3] 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

По умолчанию, создавая примитивную переменную, ей из-за примитивности данных, Java присваивает начальное значение -- ноль для числовых и ложь для булева. Что ещё раз доказывает, что мы храним там просто числа в двоичной системе счисления, мы не можем туда положить пустоту, а ноль -- это тоже значение.

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)

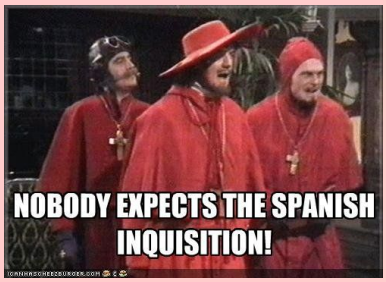
Переполнение переменных не распознаётся компилятором.

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 информации 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[4]. Для нас 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.

[4]Для более детального понимания данной темы желательно ознакомиться с информацией о цифровой схемотехнике и хранении отрицательных чисел с применением техники дополнительного кода.

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, таким образом, преобразование диапазонов из десятичной системы счисления в двоичную покажет что `byte` это один байт, `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 (три 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).

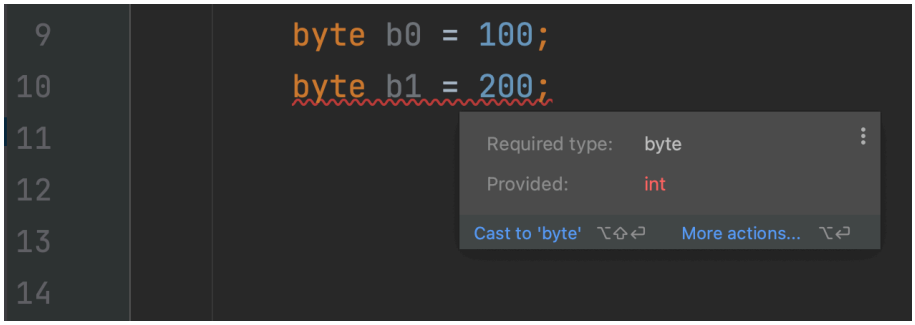


Рис. 6: Присваивание валидных и переполняющих значений

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`.

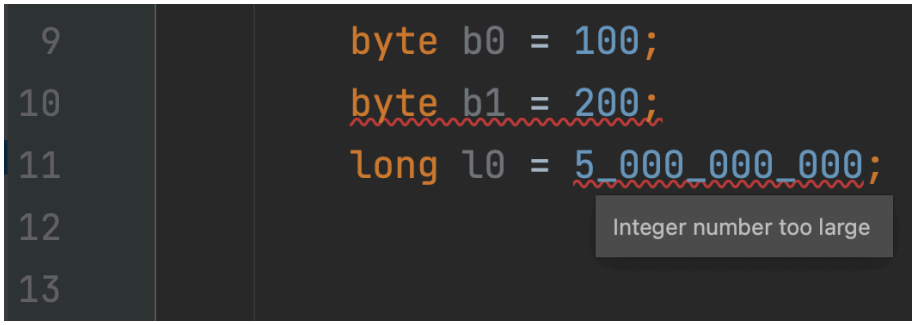


Рис. 7: Попытка инициализации переменной типа `long`

В примере на рис. 7 показана попытка присвоить значение 5000000000 (пять миллиардов) 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?

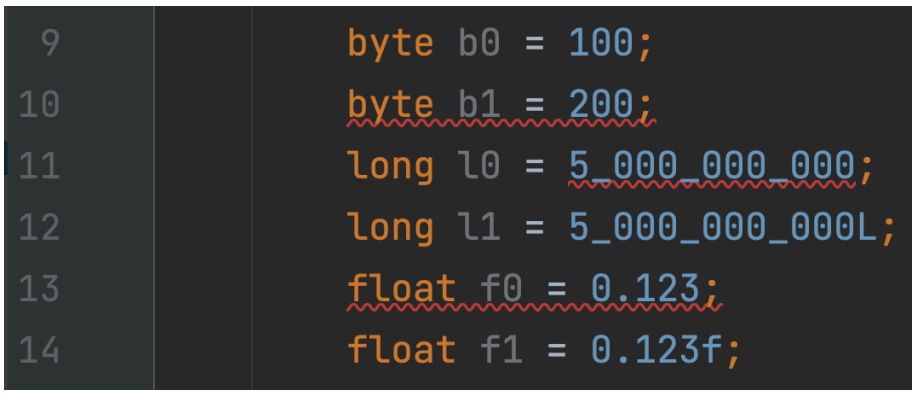


Рис. 8: Решение проблемы переполнения числовых констант

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.

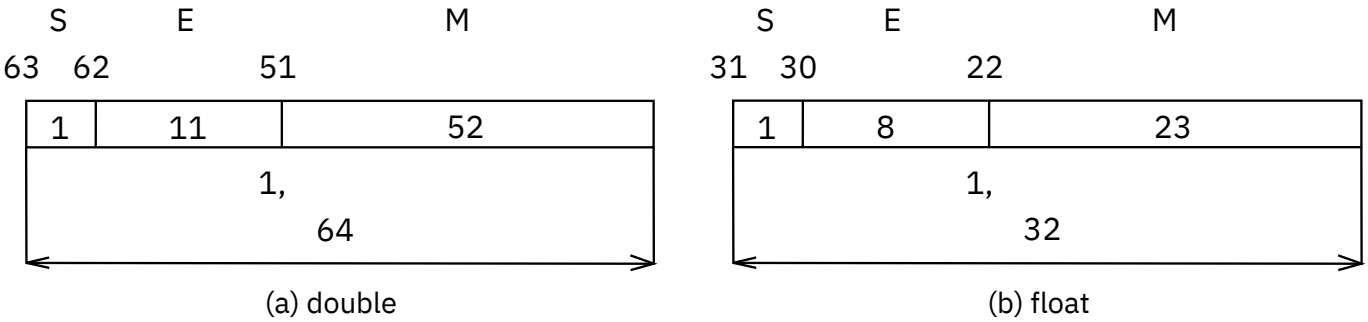
Для `long` пишем `L`, а для `float` -- `f`. Чаще всего `L` пишут заглавную, чтобы подчеркнуть, что тип больше, а `f` пишут маленькую, чтобы подчеркнуть, что мы уменьшаем тип. Но регистр в этом конкретном случае значения не имеет, можно писать и так и так.

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 `float` и `double` 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[5] 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 (или 11) разрядов поля порядка, экспонента, M -- Mantissa, 23 (или 52) бита мантиссы, дробная часть числа. Также на рисунке показана так называемая, мнимая единица, она всегда есть в самом старшем разряде мантиссы, поэтому её всегда подразумевают, но в явном виде не хранят, экономя один бит информации.

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.

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

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, но не превысить его.

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

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

$$(-1)^1 \times 2^{(124-127)} \times (1 + \frac{2097152}{2^{23}}) = -0,15625$$

или, тождественно,

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

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;

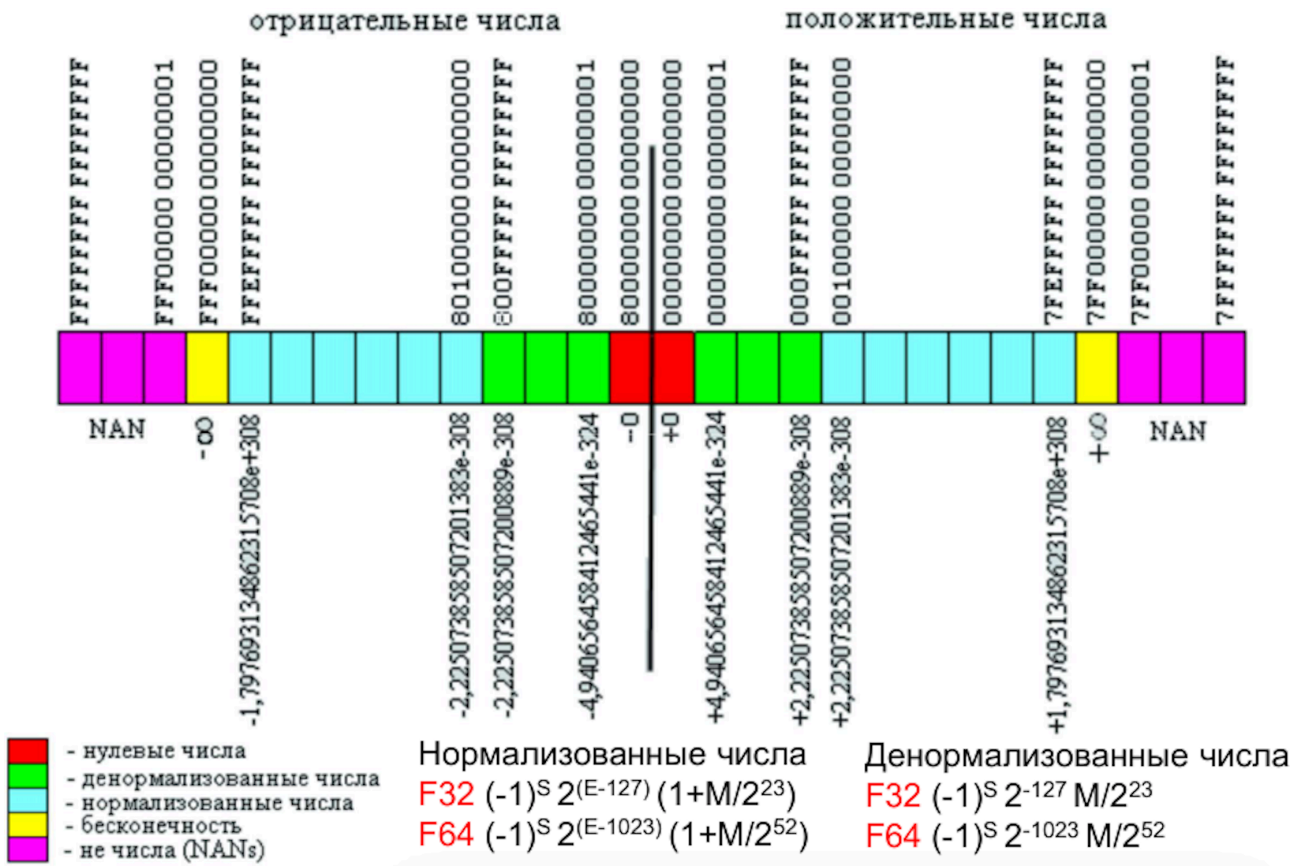


Рис. 10: Особенности работы с числами с плавающей запятой

— 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?
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

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

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 (`char`), 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 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, успешно или нет. Преобразование типов бывает явное и неявное. Неявное преобразование типов происходит когда компилятор в состоянии сам преобразовать типы, явное, когда ему нужна помощь.

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.

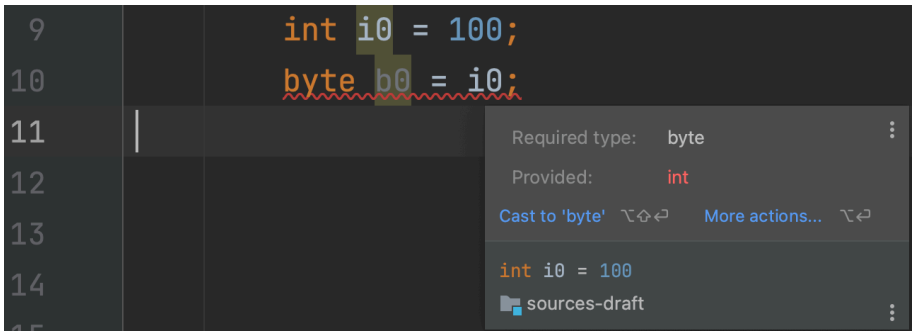


Рис. 11: Ошибка приведения типов

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.

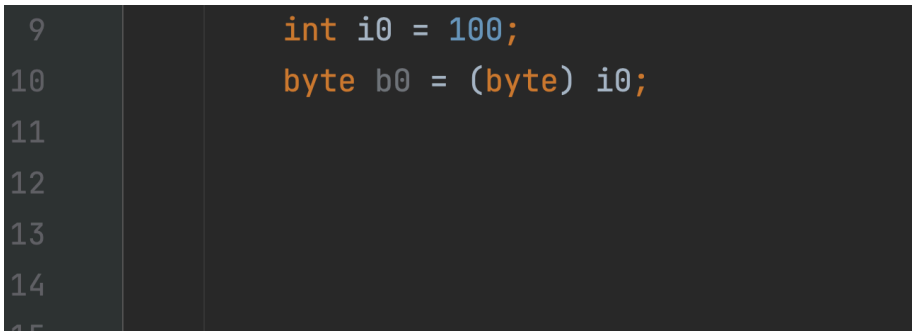


Рис. 12: Верное приведение типов

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 `final` 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}; // <-- здесь недопустимо присваивание
```

Если мы не определяем переменную, понятно, данные мы хранить не планируем. Если определяем примитивную, помним, она инициализируется нулём, а если мы определяем ссылочный идентификатор, он имеет начальное значение `null`, то есть явно демонстрирует, что не ссылается ни на какой объект. **Нулевой указатель** -- это гораздо более серьёзное явление, чем просто временное отсутствие объекта по идентификатору, очень часто это не инициализированные объекты и попытки вызова невозможных методов. Поэтому в работе очень часто помогает понять, что именно пошло не так `NullPointerException`.

Никак и никогда нельзя присвоить идентификатору целый готовый массив (создаваемый здесь и сейчас) 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 алгоритм приложения 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.

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

В каждом объекте массива есть специальное поле (рис. 13), которое обозначает длину данного массива. Поле находится в классе `__Array__` и является публичной константой.

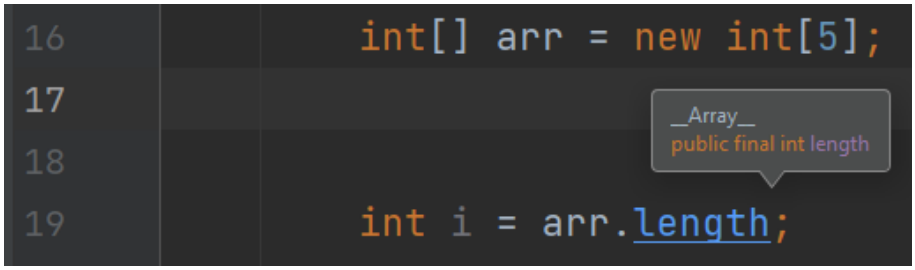


Рис. 13: Константа с длиной массива

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 Python нет. Единственное, что следует помнить, что оператор присваивания продолжает быть оператором присваивания, а не является математическим тождеством, а значит сначала посчитается всё, что слева, а потом результат попробует присвоиться переменной справа. Припоминаем какие есть особенности у операции целочисленного деления, связанные с 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 никогда 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 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 биту. Их особенность заключается в том, как язык программирования интерпретирует числа.

В Java в литеральных операциях может участвовать только тип `boolean`, в то время, как, например, 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, так или иначе, 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.

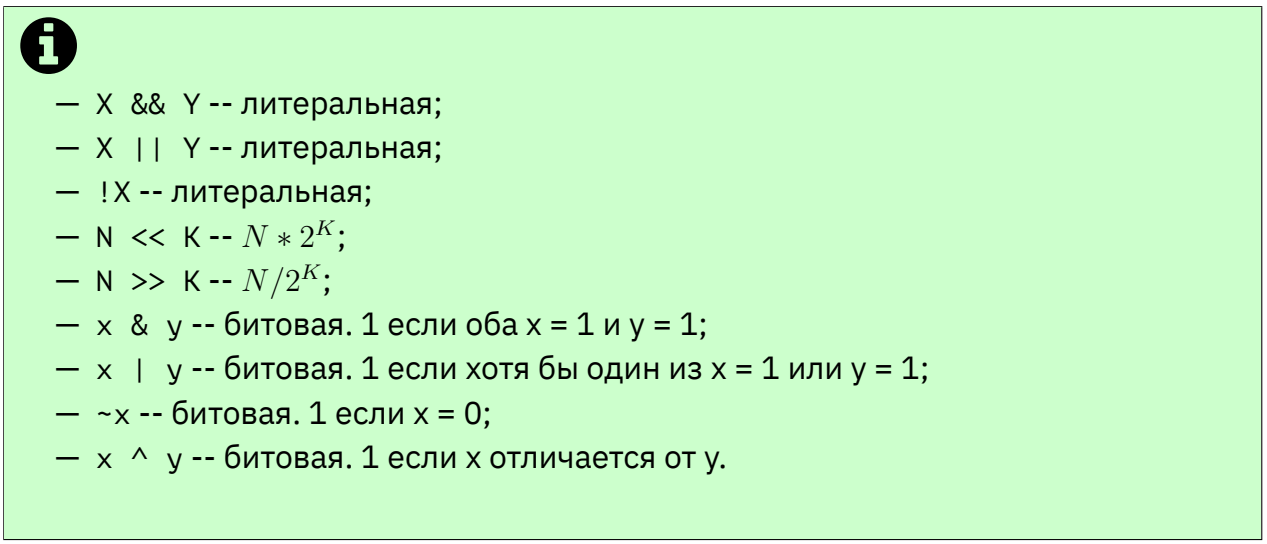
```
 00000100  4          00000100  4          00000100  4          ~00000100  4
&00000111  7         |00000111  7         ^00000111  7          11111011  -5
 00000100  4          00000111  7          00000011  3
```

Рис. 14: Бинарная арифметика

Число	Бинарное	Сдвиг	Число	Бинарное	Сдвиг
2	00000010	`2 << 0`	128	10000000	`128 >> 0`
4	00000100	`2 << 1`	64	01000000	`128 >> 1`
8	00001000	`2 << 2`	32	00100000	`128 >> 2`
16	00010000	`2 << 3`	16	00010000	`128 >> 3`
32	00100000	`2 << 4`	8	00001000	`128 >> 4`
64	01000000	`2 << 5`	4	00000100	`128 >> 5`
128	10000000	`2 << 6`	2	00000010	`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 восьмиразрядные беззнаковые 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**.

```
void method(int param1, int param2) {
   //function body
}

public static void main (String[] args) {
   method(arg1, arg2);
}
```

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.

Возвращаемого значения у метода может и не быть, в некоторых языках такие функции и методы называются процедурами, а в Java для них просто придумали специальное возвращаемое значение -- `void`. Void (от англ. пустота), интересно отметить, что это не какое-то простое emptyness, а серьёзное масштабное космическое void, чтобы подчеркнуть, что метод совсем ничего точно не возвращает.

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.

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.

Содержание

3. Специализация: ООП

3.1. В предыдущем 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).

3.2. В этом разделе

Разберём такие основополагающих в Java вещи, как классы и объекты, а также с тем, как применять на практике основные принципы ООП: наследование, полиморфизм и инкапсуляцию. Дополнительно рассмотрим устройство памяти в джава.

— Класс;
— Объект;
— Статика;
— Стек;
— Куча;
— Сборщик мусора;
— Конструктор;
— Инкапсуляция;
— Наследование;
— Полиморфизм ;

3.3. Классы и объекты, поля и методы, статика

3.3.1. Классы

Что такое класс? С точки зрения ООП, **класс** определяет форму и сущность объекта и является логической конструкцией, на основе которой построен весь язык Java.

Наиболее важная особенность класса состоит в том, что он определяет новый тип данных, которым можно воспользоваться для создания объектов этого типа

То есть класс — это шаблон (чертёж), по которому создаются объекты (экземпляры класса). Для определения формы и сущности класса указываются данные, которые он должен содержать, а также код, воздействующий на эти данные. Создаем мы свои классы, когда у нас не хватает уже созданных.

Например, если мы хотим работать в нашем приложении с документами, то необходимо для начала объяснить приложению, что такое документ, описать его в виде класса (чертежа) `Document`. Указать, какие у него должны быть свойства: название, содержание, количество страниц, информация о том, кем он подписан и т.п. В этом же классе мы обычно описываем, что можно делать с документами: печатать в консоль, подписывать, изменять содержание, название и т.д. Результатом такого описания и будет класс `Document`. Однако, это по-прежнему всего лишь чертеж хранимых данных (состояний) и способы взаимодействия с этими данными.

Если нам нужны конкретные документы, а нам они обязательно нужны, то необходимо создавать **объекты**: документ №1, документ №2, документ №3. Все эти документы будут иметь одну и ту же структуру (описанные нами название, содержание, ...), с ними можно выполнять одни и те же описанные нами действия (печатать, подписать, ...), но наполнение будет разным, например, в первом документе содержится приказ о назначении работника на должность, во втором, о выдаче премии отделу разработки и т.д.

Начнём с малого, напишем свой первый класс. Представим, что необходимо работать в приложении с котами. Java ничего не знает о том, что такое коты, поэтому необходимо создать новый класс (тип данных), и объяснить что такое кот. Создадим новый файл, для простоты в том же пакете, что и главный класс программы.

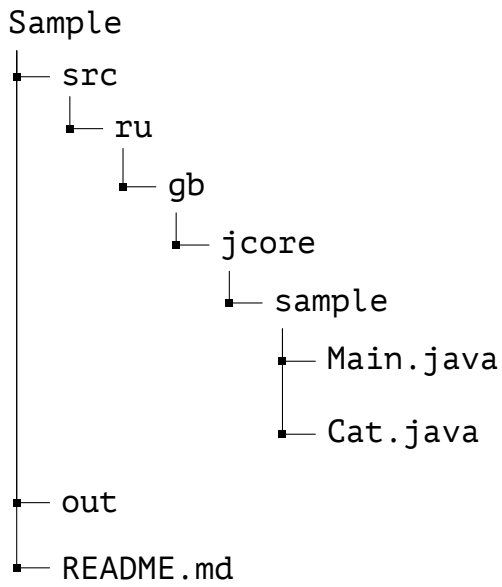


Рис. 15: Структура проекта

3.3.2. Поля класса

Начнем описывать в классе `Cat` так называемый API кота. Как известно, имя класса должно совпадать с именем файла, в котором он объявлен, т.е. класс `Cat` должен находиться в файле `Cat.java`. Пусть у котов есть три свойства: `name` (кличка), `color` (цвет) и `age` (возраст); совокупность этих свойств называется состоянием, и коты пока ничего не умеют делать. Класс `Cat` будет иметь вид, представленный в листинге 2. Свойства класса, записанные таким образом, в виде переменных, называются **полями**.

Листинг 2: Структура кота в программе

```
package ru.gb.jcore;

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

Для новичка важно не запутаться, класс кота мы описали в отдельном файле, а создавать объекты и совершать манипуляции следует в основном классе программы, не может же кот назначить имя сам себе.

Мы рассказали программе, что такое коты, теперь если мы хотим создать в нашем приложении конкретного кота, следует воспользоваться оператором `new Cat();` в основном классе программы. Более подробно разберём, что происходит в этой строке, чуть позже, пока же нам достаточно знать, что мы создали объект типа `Cat` (экземпляр класса `Cat`), и запомнить эту конструкцию. Для того чтобы с ним (экземпляром) работать, можем положить его в переменную, которой дать идентификатор `cat1`. При создании объекта полям присваиваются значения по умолчанию (нули для числовых переменных и `false` для булевых).

```
Cat cat0; // cat0 = null;
cat0 = new Cat();
Cat cat1 = new Cat();
```

В листинге выше можно увидеть все три операции (объявление, присваивание и инициализацию) и становится понятно, как можно создавать объекты. Также известно, что в переменной не лежит сам объект, а только ссылка на него. Объект `cat1` создан по чертежу `Cat`, это значит, что у него есть поля `name`, `color`, `age`, с которыми можно работать: получать или изменять их значения.

Для доступа к полям объекта используется оператор точка, который связывает имя объекта с именем поля. Например, чтобы присвоить полю `color` объекта `cat1` значение «Белый», нужно выполнить код `cat1.color = "Белый";`

Операция «точка» служит для доступа к полям и методам объекта по его имени. Мы уже использовали оператор «точка» для доступа к полю с длиной массива, например. Рассмотрим пример консольного приложения, работающего с объектами класса `Cat`. Создадим двух котов, один будет белым Барсиком 4х лет, второй чёрным Мурзиком шести лет, и просто выведем информацию о них в терминал.

```
package ru.gb.jcore;

public class Main {
   public static void main(String[] args) {
      Cat cat1 = new Cat();
```

```
6       Cat cat2 = new Cat();
7
8       cat1.name = "Barsik";
9       cat1.color = "White";
10      cat1.age = 4;
11
12      cat2.name = "Murzik";
13      cat2.color = "Black";
14      cat2.age = 6;
15
16      System.out.println("Cat1 named: " + cat1.name +
17            " is " + cat1.color +
18            " has age: " + cat1.age);
19      System.out.println("Cat2 named: " + cat2.name +
20            " is " + cat2.color +
21            " has age: " + cat2.age);
22  }
23 }
```

в результате работы программы в консоли появятся следующие строки:

```
Cat1 named: Barsik is White has age: 4
Cat2 named: Murzik is Black has age: 6
```

Вначале мы создали два объекта типа `Cat`: `cat1` и `cat2`, соответственно, они имеют одинаковый набор полей `name`, `color`, `age`. Почему? Потому что они принадлежат одному классу, созданы по одному шаблону. Объекты всегда «знают», какого они класса. Однако каждому из них в эти поля записаны разные значения. Как видно из результата печати в консоли, изменение значения полей одного объекта, никак не влияет на значения полей другого объекта. Данные объектов `cat1` и `cat2` изолированы друг от друга. А значит мы делаем вывод о том, поля хранятся в классе, а значения полей хранятся в объектах. Логическая структура, демонстрирующая отношения объектов и классов, в том числе в части хранения полей и их значений показана на рис. 16.

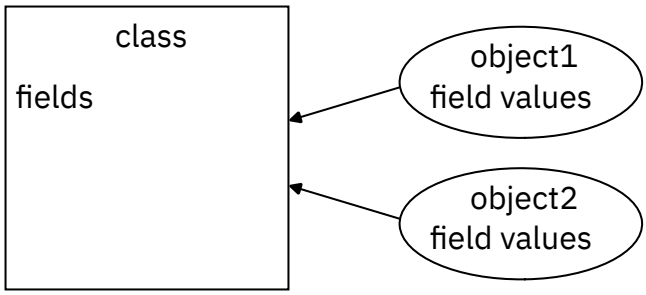


Рис. 16: Логическая структура отношения класс-объект

3.3.3. Объекты

Разобравшись с тем, как создавать новые типы данных (классы) и мельком посмотрев, как создаются объекты, нужно подробнее разобрать, как создавать объекты, и что при этом происходит. Создание объекта как любого ссылочного типа данных проходит в два этапа. Как и в случае с уже известными нам массивами.

— Сначала создается переменная, имеющая интересующий нас тип, в неё возможно записать ссылку на объект;
— затем необходимо выделить память под объект;

— создать и положить объект в выделенную часть памяти;
— и сохранить ссылку на этот объект в памяти -- в переменную.

Для непосредственного создания объекта применяется оператор new, который динамически резервирует память под объект и возвращает ссылку на него, в общих чертах эта ссылка представляет собой адрес объекта в памяти, зарезервированной оператором new.

```
Cat cat1; // cat1 = null;
cat1 = new Cat();
Cat cat2 = new Cat();
```

В первой строке кода переменная cat1 объявляется как ссылка на объект типа Cat и пока ещё не ссылается на конкретный объект (первоначально значение переменной cat1 равно null). В следующей строке выделяется память для объекта типа Cat, и в переменную cat1 сохраняется ссылка на него. После выполнения второй строки кода переменную cat1 можно использовать так, как если бы она была объектом типа Cat. Обычно новый объект создается в одну строку, то есть инициализируется.

3.3.4. Оператор new

[квалификаторы] ИмяКласса имяПеременной = **new** ИмяКласса();

Оператор new динамически выделяет память для нового объекта, общая форма применения этого оператора имеет вид как на врезке выше, но на самом деле справа -- не имя класса, конструкция ИмяКласса() в правой части выполняет вызов конструктора данного класса, который подготавливает вновь создаваемый объект к работе.

Именно от количества применений оператора new будет зависеть, сколько именно объектов будет создано в программе.

```
Cat cat1 = new Cat();
Cat cat2 = cat1;

cat1.name = "Barsik";
cat1.color = "White";
cat1.age = 4;

cat2.name = "Murzik";
cat2.color = "Black";
cat2.age = 6;

System.out.println("Cat1 named: " + cat1.name +
      " is " + cat1.color +
      " has age: " + cat1.age);
System.out.println("Cat2 named: " + cat2.name +
      " is " + cat2.color +
      " has age: " + cat2.age);
```

На первый взгляд может показаться, что переменной cat2 присваивается ссылка на копию объекта cat1, т.е. переменные cat1 и cat2 будут ссылаться на разные объекты в памяти. Но это не так. На самом деле cat1 и cat2 будут ссылаться на один и тот же объект.

Присваивание переменной `cat1` значения переменной `cat2` не привело к выделению области памяти или копированию объекта, лишь к тому, что переменная `cat2` содержит ссылку на тот же объект, что и переменная `cat1`. Это явление дополнительно подчёркивает ссылочную природу данных в языке Java.

Таким образом, любые изменения, внесённые в объект по ссылке `cat2`, окажут влияние на объект, на который ссылается переменная `cat1`, поскольку *это один и тот же объект в памяти*. Поэтому результатом выполнения кода, где мы как будто бы указали возраст второго кота, равный шести годам, станут строки, показывающие, что по обеим ссылкам оказался кот возраста шесть лет с именем Мурзика.

```
Cat1 named: Murzik is Black has age: 6
Cat2 named: Murzik is Black has age: 6
```

Множественные ссылки на один и тот же объект в памяти довольно легко себе представить как ярлыки для запуска одной и той же программы на рабочем столе и в меню быстрого запуска. Или если на один и тот же шкафчик в раздевалке наклеить два номера -- сам шкафчик можно будет найти по двум ссылкам на него.

Важно всегда перепроверять, какие объекты созданы, а какие имеют множественные ссылки.

3.3.5. Методы

Ранее было сказано о том, что в языке Java любая программа состоит из классов и функций, которые могут описываться только внутри них. Именно поэтому все функции в языке Java являются методами. А метод -- это функция, являющаяся частью некоторого класса, которая может выполнять операции над данными этого класса.

Метод -- это функция, принадлежащая классу

Метод для своей работы может использовать поля объекта и/или класса, в котором определен, напрямую, без необходимости передавать их во входных параметрах. Это похоже на использование глобальных переменных в функциях, но в отличие от глобальных переменных, метод может получать прямой доступ только к членам класса. Самые простые методы работают с данными объектов. Методы чаще всего формируют API классов, то есть способ взаимодействия с классами, интерфейс. Место методов во взаимодействии классов и объектов показано на рис. 17.

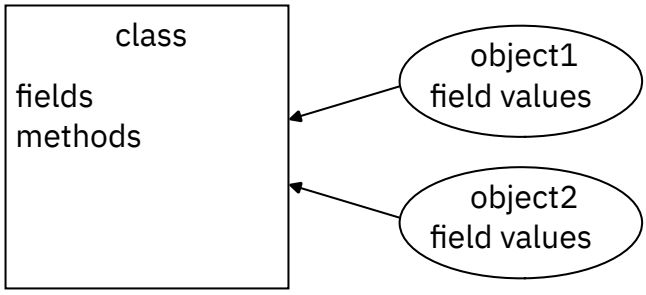


Рис. 17: Логическая структура отношения класс-объект

Вернёмся к примеру с котиками. Широко известно, что котики умеют урчать, мяукать и смешно прыгать. В целях демонстрации в описании этих действий просто будем делать разные выводы в консоль, хотя возможно и научить котика в программе выбирать минимальное значение из массива, но это было бы, как минимум, неожиданно. Итак опишем метод например подать голос и прыгать.

```
public class Cat {
   String name;
   String color;
   int age;

   void voice() {
      System.out.println(name + " meows");
   }

   void jump() {
      if (this.age < 5) System.out.println(name + " jumps");
   }
}
```

Обращение к методам выглядит очень похожим на стандартный способом, через точку, как к полям. Теперь когда появляется необходимость позвать котика, он скажет: «мяу, я имя котика», а если в программе пришло время котику прыгнуть, он решит, прилично ли это -- прыгать в его возрасте.

```
package ru.gb.jcore;

public class Main {
   public static void main(String[] args) {
      Cat cat1 = new Cat();
      Cat cat2 = new Cat();

      cat1.name = "Barsik";
      cat1.color = "White";
      cat1.age = 4;

      cat2.name = "Murzik";
      cat2.color = "Black";
      cat2.age = 6;

      cat1.voice();
      cat2.voice();
      cat1.jump();
      cat2.jump();
   }
}
```

```
Barsik meows
Murzik meows
Barsik jumps
```

Как видно, Барсик замечательно прыгает, а Мурзик от прыжков воздержался, хотя попрыгать программа попросила их обоих.

3.3.6. Ключевое слово `static`

В завершение базовой информации о классах и объектах, остановимся на специальном модификаторе `static`, делающем переменную или метод «независимыми» от объекта.

`static` — модификатор, применяемый к полю, блоку, методу или внутреннему классу, он указывает на привязку субъекта к текущему классу.

Для использования таких полей и методов, соответственно, объект создавать не нужно. В Java большинство членов служебных классов являются статическими. Возможно использовать это ключевое слово в четырех контекстах:

— статические методы;
— статические переменные;
— статические вложенные классы;
— статические блоки.

В этом разделе рассмотрим подробнее только первые два пункта, третий опишем чуть позже, а четвёртый потребует от нас знаний, выходящих не только за этот урок, но и за десяток следующих.

Статические методы также называются методами класса, потому что статический метод принадлежит классу, а не его объекту. Нестатические называются методами объекта. Статические методы можно вызывать напрямую через имя класса, не обращаясь к объекту и вообще объект не создавая. Что это и зачем нужно? Например, умение кота мяукать можно вывести в статическое поле, потому что, например, весной можно открыть окно, не увидеть ни одного экземпляра котов, но зато услышать их, и точно знать, что мяукают не дома и не машины, а именно коты.

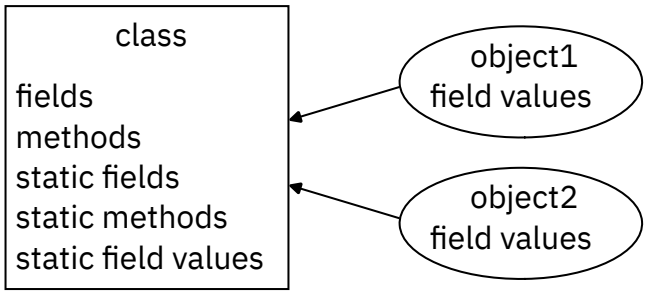


Рис. 18: Логическая структура отношения класс-объект

Аналогично статическим методам, **статические поля** принадлежат классу и совершенно ничего «не знают» об объектах.

Важной отличительной чертой статических полей является то, что их значения также хранятся в классе, в отличие от обычных полей, чьи значения хранятся в объектах.

Рисунок 18 именно в этом виде автор настоятельно рекомендует если не заучить, то хотя бы хорошо запомнить, он ещё пригодится в дальнейшем обучении и работе. Из этого же изображения можно сделать несколько выводов.

```java
public class Cat {
   static int pawsCount = 4;

   String name;
   String color;
   int age;

   // ...
}
```

Помимо того, что статические поля -- это полезный инструмент создания общих свойств это ещ§ и опасный инструмент создания общих свойств. Так, например, мы знаем, что у котов четыре лапы, а не 6 и не 8. Не создавая никакого барсика будет понятно, что у кота -- 4 лапы. Это полезное поведение.

лайвкод 03-статическое-поле-ошибка Посмотрим на опасность. Мы видим, что у каждого кота есть имя, и помним, что коты хранят значение своего имени каждый сам у себя. А знают экземпляры о названии поля потому что знают, какого класса они экземпляры. Но что если мы по невнимательности добавим свойство статичности к имени кота?

03-статическое-поле-признак Создав тех же самых котов, которых мы создавали весь урок, мы получим двух мурзиков и ни одного барсика. Почему это произошло? По факту переменная у нас одна на всех, и значение тоже одно, а значит каждый раз мы меняем именно его, а все остальные коты ничего не подозревая смотрят на значение общей переменной и бодро его возвращают. Поэтому, чтобы не запутаться, к статическим переменным, как правило, обращаются не по ссылке на объект — cat1.name, а по имени класса — Cat.name.

03-статические-поля К слову, статические переменные — редкость в Java. Вместо них применяют статические константы. Они определяются ключевыми словами static final и по соглашению о внешнем виде кода пишутся в верхнем регистре.

3.3.7. Задание для самопроверки

1. Что такое класс?
2. Что такое поле класса?
3. На какие три этапа делится создание объекта?
4. Какое свойство добавляет ключевое слово static полю или методу?
   (a) неизменяемость;
   (b) принадлежность классу;
   (c) принадлежность приложению.
5. Может ли статический метод получить доступ к полям объекта?
   (a) не может;
   (b) может только к константным;
   (c) может только к неинициализированным.

3.4. Устройство памяти. Стек, куча и сборка мусора

Это погружение в управление памятью Java позволит расширить ваши знания о том, как работает куча, ссылочные типы и сборка мусора, понять глубинные процессы и, как следствие, писать более хорошие программы. Для оптимальной работы приложения JVM

делит память на область стека (stack) и область кучи (heap). Всякий раз, когда объявляются новые переменные, создаются объекты или вызывается новый метод, JVM выделяет память для этих операций в стеке или в куче. На рисунке 19 представлена общая модель организации памяти в Java.

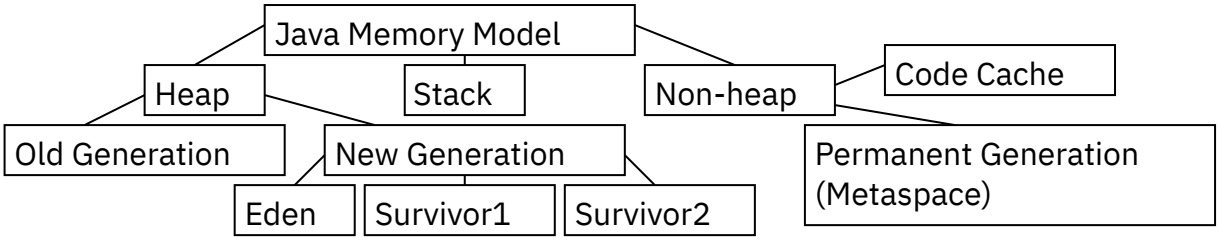


Рис. 19: Устройство памяти

Стековая память отвечает за хранение ссылок на объекты кучи и за хранение типов значений (также известных в Java как примитивные типы), которые содержат само значение, а не ссылку на объект из кучи. Данная память в Java работает по схеме LIFO (last in, first out, последний зашел, первый вышел).

Немного забегая вперёд можно сказать, что все потоки, работающие в JVM, имеют свой стек. Пока достаточно отождествлять поток и собственно исполнение программы. Стек в свою очередь держит информацию о том, какие методы вызвал поток. Назовём это «стеком вызовов». Стек вызовов возобновляется, как только поток завершает выполнение своего кода. Каждый слой стека вызовов содержит все локальные переменные для вызываемого метода и потока. Все локальные переменные примитивных типов полностью хранятся в стеке потоков и не видны другим потокам.

Особенности стека:

— Он заполняется и освобождается по мере вызова и завершения новых методов;
— Переменные на стеке существуют до тех пор, пока выполняется метод в котором они были созданы;
— Если память стека будет заполнена, Java бросит исключение `java.lang.StackOverflowError`;
— Доступ к этой области памяти осуществляется быстрее, чем к куче;
— Является потокобезопасным, поскольку для каждого потока создается свой отдельный стек.

Куча содержит все объекты, созданные в приложении, независимо от того, какой поток создал объект. Неважно, был ли объект создан и присвоен локальной переменной или создан как переменная-член другого объекта, он хранится в куче.

Локальная переменная может быть примитивной, но также может быть ссылкой на объект. В этом случае ссылка (локальная переменная) хранится на стеке, но сам объект хранится в куче. Объект использует методы, эти методы содержат локальные переменные. Эти локальные переменные (то есть в момент выполнения метода) также хранятся на стеке, несмотря на то, что объект, который использует метод, хранится в куче. Переменные-члены класса хранятся в куче вместе с самим классом. Это верно как в случае, когда переменная-член имеет примитивный тип, так и в том случае, если она является ссылкой на объект. Статические переменные класса также хранятся в куче вместе с определением класса.

В общем случае, эти объекты имеют глобальный доступ и могут быть получены из любого места программы.

Куча разбита на несколько более мелких частей, называемых поколениями:

— Young Generation — область где размещаются недавно созданные объекты. Когда она заполняется, происходит быстрая сборка мусора;
— Old (Tenured) Generation — здесь хранятся долгоживущие объекты. Когда объекты из Young Generation достигают определенного порога «возраста», они перемещаются в Old Generation;
— Permanent Generation — эта область содержит метаинформацию о классах и методах приложения, но начиная с Java 8 данная область памяти была упразднена. В Java 8 Permanent Generation заменён на Metaspace -- его динамически изменяемый по размеру аналог. Именно здесь находятся статические поля.

Особенности кучи:

— В общем случае, размеры кучи на порядок больше размеров стека
— Когда эта область памяти полностью заполняется, Java бросает `java.lang.OutOfMemoryError`;
— Доступ к ней медленнее, чем к стеку;
— Эта память, в отличие от стека, автоматически не освобождается. Для сбора неиспользуемых объектов используется сборщик мусора;
— В отличие от стека, который создаётся для каждого потока свой, куча не является потокобезопасной, поскольку для всех одна, и ее необходимо контролировать, правильно синхронизируя код.

Также, хотелось бы отметить, что мы можем использовать -Xms и -Xmx опции JVM, чтобы определить начальный и максимальный размер памяти в куче. Для стека определить размер памяти можно с помощью опции -Xss;

Управление неиспользуемыми объектами

— запускается автоматически Java, и Java решает, запускать или нет этот процесс;
— это дорогостоящий процесс. При запуске сборщика мусора все потоки в приложении приостанавливаются (в зависимости от типа GC);
— гораздо более сложный процесс, чем просто сбор мусора и освобождение памяти.

Программисту доступен метод класса System, который мы можем явно вызвать и ожидать, что сборщик мусора будет запускаться при выполнении этой строки кода. Это ошибочное предположение. Java решать, делать это или нет. Явно вызывать `System.gc()` не рекомендуется.

Поскольку это довольно сложный процесс и может повлиять на производительность всего приложения, он реализован весьма разумно. Для этого используется так называемый процесс «Mark and Sweep» (отмечай и подметай). Java анализирует переменные из

стека и «отмечает» все объекты, которые необходимо поддерживать в рабочем состоянии. Затем все неиспользуемые объекты очищаются. Фактически, чем больше мусора и чем меньше объектов помечены как живые, тем быстрее идет процесс. Чтобы сделать это еще более оптимизированным, память кучи состоит из нескольких частей.

1. Молодое поколение -- Все новые объекты начинаются с молодого поколения. Как только они выделены в коде Java, они попадают в этот подраздел, называемый **eden space**. В конце концов пространство эдема заполняется объектами. На этом этапе происходит незначительная сборка мусора, так называемая minor collection. Некоторые объекты (те, на которые есть ссылки) помечаются, а некоторые (те, на которые нет ссылок) -- нет. Те, которые были отмечены, затем переходят в другой подраздел молодого поколения под названием пространство выживших (само пространство выживших разделено на две части). Те, которые остались немаркированными, удаляются автоматической сборкой мусора.
2. Выжившее поколение -- Так будет продолжаться до тех пор, пока пространство eden снова не заполнится; на этом этапе начинается новый цикл. События minor collection повторяются, но в этом цикле все отмеченные объекты, которые выживают как из пространства eden, так и из S0, фактически попадают во вторую часть пространства survivor, называемую S1.
3. Третье поколение -- Любые объекты, попадающие в пространство выживших, помечаются счетчиком возраста. Алгоритм проверяет этот счётчик, чтобы увидеть, соответствует ли он пороговому значению для перехода в старое поколение. Главная мысль в том, что объекты не обязательно переходят из S0 в S1 пространства выживших. На самом деле, они просто чередуются с тем, куда они переключаются при каждой minor сборке мусора.

   Если эти процессы обобщить, то все новые объекты начинаются в пространстве eden, а затем в конечном итоге попадают в пространство survivor, поскольку они переживают несколько циклов сборки мусора.
4. Старое поколение можно рассматривать как место, где лежат долгоживущие объекты. Когда объекты собирают мусор из старого поколения, происходит крупное событие сборки мусора. Старое поколение состоит только из одной секции, называемой постоянным поколением.
5. Постоянное поколение -- Постоянное поколение не заполняется, когда объекты старого поколения достигают определенного порога, а затем перемещаются (повышаются) в постоянное поколение. Скорее, постоянное поколение немедленно заполняется JVM метаданными, которые представляют классы и методы приложений во время выполнения. JVM иногда может следовать определенным правилам для очистки постоянного поколения, и когда это происходит, это называется полной сборкой мусора major collection.

 Также, хотелось бы ещё раз упомянуть событие под названием остановить мир. Когда происходит небольшая сборка мусора (для молодого поколения) или крупная сборка мусора (для старого поколения), мир останавливается; другими словами, все потоки приложений полностью останавливаются и должны ждать завершения события сборки мусора.

Сборщик мусора. Реализации

1. Последовательный сборщик мусора. Это самая простая реализация GC, поскольку она в основном работает с одним потоком. В результате эта реализация GC замораживает все потоки приложения при запуске. Поэтому не рекомендуется использовать его в многопоточных приложениях, таких как серверные среды;
2. Параллельный сборщик мусора. Это GC по умолчанию для JVM, который иногда называют сборщиками пропускной способности. В отличие от последовательного сборщика мусора, он использует несколько потоков для управления пространством кучи, но также замораживает другие потоки приложений во время выполнения GC. Если мы используем этот GC, мы можем указать максимальные потоки сборки мусора и время паузы, пропускную способность и занимаемую площадь (размер кучи);
3. Сборщик мусора CMS. Реализация Concurrent Mark Sweep (CMS) использует несколько потоков сборщика мусора для сбора мусора. Он предназначен для приложений, которые требуют более коротких пауз при сборке мусора и могут позволить себе совместно использовать ресурсы процессора со сборщиком мусора во время работы приложения. Проще говоря, приложения, использующие этот тип GC, в среднем работают медленнее, но не перестают отвечать, чтобы выполнить сборку мусора.

 Следует отметить, что, поскольку этот GC является параллельным, вызов явной сборки мусора, такой как использование System.gc() во время работы параллельного процесса, приведет к сбою или прерыванию параллельного режима;

4. Сборщик мусора G1. Сборщик мусора G1 (Garbage First) предназначен для приложений, работающих на многопроцессорных компьютерах с большим объемом памяти. Он доступен с обновления 4 JDK7 и в более поздних версиях. Сборщик G1 заменит сборщик CMS, поскольку он более эффективен;
5. Z сборщик мусора. ZGC (Z Garbage Collector) -- это масштабируемый сборщик мусора с низкой задержкой, который дебютировал в Java 11 в качестве экспериментального варианта для Linux. JDK 14 представил ZGC под операционными системами Windows и macOS. ZGC получил статус production начиная с Java 15.

Итоги рассмотрения устройства памяти

— куча доступна везде, объекты доступны отовсюду
— все объекты хранятся в куче, все локальные переменные хранятся на стеке
— стек недолговечен
— и стек и куча могут быть переполнены
— куча много больше стека, но стек гораздо быстрее

3.4.1. Задания для самопроверки

1. По какому принципу работает стек?
2. Что быстрее, стек или куча?
3. Что больше, стек или куча?

3.5. Конструкторы

3.5.1. Контроль над созданием объекта

Чтобы создать объект мы тратим одну строку кода `Cat cat1 = new Cat();` поля этого объекта заполнятся автоматически значениями по-умолчанию (числовые -- 0, логические -- `false`, ссылочные -- `null`). Часто нужно при создании дать коту какое-то имя, указать его возраст и цвет, поэтому пишем ещё три строки кода.

В таком подходе есть несколько недостатков:

1. прямое обращение к полям объекта,
2. если полей у класса будет намного больше, то для создания всего лишь одного объекта будет уходить 5-10-15 строк кода, что очень громоздко и утомительно.

Было бы неплохо иметь возможность сразу, при создании объекта указывать значения его полей. Для инициализации объектов при создании в Java предназначены конструкторы.

Конструктор -- это частный случай метода в том смысле, что он тоже выполняет какие-то действия. Имя конструктора обязательно должно совпадать с именем класса, возвращаемое значение не пишется.

Если создать конструктор класса Cat, как показано в листинге 3, он автоматически будет вызываться при создании объекта. Теперь, при создании объектов класса Cat, все коты будут иметь одинаковые имена, цвет и возраст (это будут белые двухлетние Барсики).

Листинг 3: Не самый лучший конструктор

```
1  public class Cat {
2     String name;
3     String color;
4     int age;
5
6     public Cat() {
7        name = "Barsik";
8        color = "White";
9        age = 2;
10    }
11
12    // ...
13 }
```

При использовании такого конструктора, все созданные коты будут одинаковыми, пока мы вручную не поменяем значения их полей. Чтобы можно было указывать начальные значения полей объектов необходимо создать параметризированный конструктор.

Листинг 4: Параметризированный конструктор

```
public class Cat {
   String name;
   String color;
   int age;

   Cat(String n, String c, int a) {
      name = n;
      color = c;
      age = a;
   }

   // ...
}
```

В приведенном примере, в параметрах конструктора используется первая буква от названия поля, это сделано для упрощения понимания логики заполнения полей объекта, и будет заменено на более корректное использование ключевого слова `this`. При такой форме конструктора, когда появится необходимость создавать в программе кота, необходимо будет обязательно указать его имя, цвет и возраст. Набор полей, которые будут заполнены через конструктор, определяется разработчиком класса сами, то есть язык не обязывает заполнять все поля, которые есть в классе записывать в параметры конструктора, но при вызове обязательно заполнить аргументами все, что есть в параметрах, как при вызове метода.

```
Cat cat1 = new Cat("Barsik", "White", 4);
Cat cat2 = new Cat("Murzik", "Black", 6);
```

Наборы значений имён, цветов и возрастов будут переданы в качестве аргументов конструктора (`n`, `c`, `a`), а конструктор уже перезапишет полученные значения в поля объект (`name`, `color`, `age`). То есть начальные значения полей каждого из объектов будут определяться тем, что мы передадим ему в конструкторе.

Язык позволяет как не объявлять ни одного конструктора, так и объявить их несколько. Также как и при перегрузке методов, имеет значение набор аргументов, не может быть нескольких конструкторов с одинаковым набором аргументов. Соответствующий перегружаемый конструктор вызывается в зависимости от аргументов, указываемых при выполнении оператора new.

Как только в программе в классе создана своя реализация конструктора, пустой конструктор, называемый также **конструктором по-умолчанию** автоматически создаваться не будет. И если понадобится такая форма конструктора, необходимо будет создать его вручную, `public Cat() {}`.

То есть, компилятор как-бы думает, что если вы не описали конструкторы, значит вам не важно, как будет создаваться объект, а значит, хватит пустого конструктора, ну а если конструктор написан, значит выкручивайтесь сами.

3.5.2. Ключевое слово `this`

В контексте конструкторов, применять this нужно в двух случаях:

1. Когда у переменной экземпляра класса и переменной метода/конструктора одинаковые имена;
2. Когда нужно вызвать конструктор одного типа (например, конструктор по умолчанию или параметризированный) из другого. Это еще называется явным вызовом конструктора.

Внимательно посмотрев на параметризированный конструктор (листинг 4), видим, что переменные в параметрах называются не также, как поля класса.

Нельзя просто сделать названия параметров идентичными названиям полей, в этом случае возникает проблема. Для примера возьмём имя кота, поле `String name`. Один `String name` принадлежит классу `Cat`, а другой `String name` находится в локальной видимости конструктора. JVM, как и любой другой электрический прибор всегда идёт по пути наименьшего сопротивления, когда есть неопределённость. То есть, когда написана строка `name = name;` Java берёт самую близкую `name` из конструктора и для левой и для правой части оператора присваивания, что не имеет никакого смысла.

Листинг 5: Использование ключевого слова `this` для параметров

```
1  public class Cat {
2     String name;
3     String color;
4     int age;
5
6     public Cat(String name, String color, int age) {
7        this.name = name;
8        this.color = color;
9        this.age = age;
10    }
11
12    // ...
13 }
```

Ключевое слово `this` сошлётся на вызвавший объект, в результате чего (в листинге 6)имя котика через конструктор будет установлено создаваемому объекту. Таким образом, здесь `this` позволяет не вводить новые переменные для обозначения одного и того же, что позволяет сделать код менее перегруженным дополнительными переменными.

Второй случай частого использования `this` с конструкторами -- вызов одного конструктора из другого. это может пригодиться когда в классе описано несколько конструкторов и

не хочется в новом конструкторе переписывать код инициализации, приведенный в конструкторе ранее[6]. В листинге 6 вызывается обычный конструктор с тремя параметрами, который принимает имя цвет и возраст, но, допустим, когда котята рождаются возраст им задавать смысла нет, поэтому, может пригодиться и конструктор просто с именем и цветом, а зачем писать присваивание имени и цвета несколько раз, если можно вызвать соответствующий конструктор?

Листинг 6: Использование ключевого слова `this` для параметров

```
public class Cat {
   String name;
   String color;
   int age;

   public Cat(String name, String color) {
      this.name = name;
      this.color = color;
   }

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

   // ...
}
```

На такой вызов есть ограничение, конструктор из конструктора можно вызвать только один раз и только на первой строке конструктора.

Ключевое слово `this` в Java используется только в составе экземпляра класса. Но неявно ключевое слово `this` передается во все методы, кроме статических (поэтому `this` часто называют неявным параметром) и может быть использовано для обращения к объекту, вызвавшему метод.

Существует ещё один вид конструктора -- это **конструктор копирования**. Чтобы создать конструктор копирования, возможно объявить конструктор, который принимает объект того же типа, в нашем случае котика, в качестве параметра, а в самом конструкторе аналогично конструктору, заполняющему все параметры, заполнить каждое поле входного объекта в новый экземпляр.

```
public Cat (Cat cat) {
   this(cat.name, cat.color, cat.age);
}
```

Благодаря имеющемуся конструктору со всеми нужными параметрами, с помощью ключевого слова `this` явно вызывается конструктор заполняющий все поля создаваемого кота, значениями из переданного объекта, фактически, его копирующий. То, что мы имеем

[6] один из базовых принципов программирования -- DRY (от англ dry -- чистый, сухой, акроним don't repeat yourself) -- не повторяйся. Его антагонист WET (от англ wet -- влажный, акроним write everything twice) -- пиши всё дважды.

здесь, – это неглубокая копия. Если класс имеет изменяемые поля, например, массивы, то мы можем вместо простой сделать глубокую копию внутри его конструктора копирования. При глубокой копии вновь созданный объект не должен зависеть от исходного, а значит просто скопировать ссылку на массив будет недостаточно

3.5.3. Задания для самопроверки

1. Для инициализации нового объекта абсолютно идентичными значениями свойств переданного объекта используется
   (a) пустой конструктор
   (b) конструктор по-умолчанию
   (c) конструктор копирования
2. Что означает ключевое слово this?

3.6. Инкапсуляция

Инкапсуляция связывает данные с манипулирующим ими кодом и позволяет управлять доступом к членам класса из отдельных частей программы, предоставляя доступ только с помощью определенного ряда методов, что позволяет предотвратить злоупотребление этими данными. То есть класс должен представлять собой «черный ящик», которым возможно пользоваться, но его внутренний механизм защищен от повреждений.

Инкапсуляция -- (англ. encapsulation, от лат. in capsula) -- в информатике, процесс разделения элементов абстракций, определяющих ее структуру (данные) и поведение (методы); инкапсуляция предназначена для изоляции контрактных обязательств абстракции (протокол/интерфейс) от их реализации.

В Java в роли чёрного ящика выступает класс. Класс содержит в себе и данные (поля класса), и действия (методы класса) для работы с этими данными. Все члены класса в языке Java -- поля и методы -- имеют модификаторы доступа. Ранее уже было описан модификатор `public`, означающий доступность отовсюду, обычно используется для методов.

Модификаторы доступа позволяют задать допустимую область видимости для членов класса, то есть контекст, в котором можно употреблять данную переменную или метод.

Есть два модификатора, которые уже известны и активно используются, это `public` и package-private, также известный как default, пакетный или отсутствующий модификатор. Что это значит? Это значит, что вообще всё что пишется в Java имеет уровень доступа, и если этот уровень не определён явно, то Java отнесёт данные к уровню доступности внутри пакета.

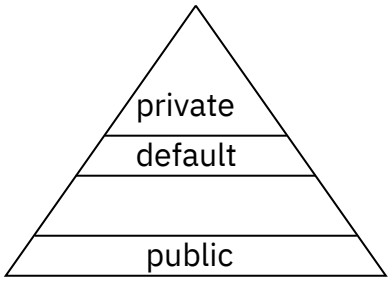


Рис. 20: Модификаторы доступа и их относительная область видимости

Модификатор `private` определяет доступность только внутри класса, и предпочтительнее всех.

Внимательно рассмотрим класс котика из листинга 6. Например, кто-то может создать хорошего кота, а потом его переименовать, перекрасить в зелёный цвет или сделать ему отрицательный возраст, в результате в программе находятся объекты с некорректным состоянием. Все поля находятся в пакетном доступе. К ним можно обратиться в любом месте пакета: достаточно просто создать объект.

`private` — самый строгий модификатор доступа в Java. Если его использовать, поля класса не будут доступны за его пределами. Решая проблему несанкционированного доступа была получена проблема штатного функционирования, доступ к полям закрыт, в программе нельзя даже получить вес существующей кошки, если это понадобится.

Необходимо решить вопросы с получением и изменением значений полей. На помощь приходят «геттеры» и «сеттеры». Название происходит от английского «get» — «получать» (т.е. «метод для получения значения поля») и «set» — «устанавливать».

Листинг 7: Геттеры и сеттеры для всех полей

```
public class Cat {
   private String name;
   private String color;
   private int age;

   // ...

   public String getName() {
     return name;
   }
   public String getColor() {
     return color;
   }
   public int getAge() {
     return age;
   }
   public void setName(String name) {
     this.name = name;
   }
   public void setColor(String color) {
     this.color = color;
   }
   public void setAge(int age) {
     this.age = age;
   }
}
```

В листинге 7 показан пример написания геттеров и сеттеров для всех полей класса, что даёт возможность получать данные и устанавливать их. Например, с помощью `getColor` возможно получить текущее значение окраса котика.

Важно, что создавая для класса геттеры и сеттеры не только появляется возможность дополнять установке и возвращению значений полей дополнительную логику, но и возможность регулировать доступ к полям. Например, если в программе нужно запретить менять котикам окрас, то для класса просто не пишется соответствующий сеттер.

Внимательно осмотрев класс кота возможно прийти к выводу, что хранить возраст котов очень неудобно, потому что каждый год нужно будет обновлять это значение для каждого объекта кота в программе, а это может оказаться утомительно. Выходом может оказаться хранение не возраста, а неизменяемого параметра -- даты рождения и подсчёт возраста каждый раз, когда его запрашивают, ведь человеку, который запрашивает возраст кота, не интересно, каким образом получено значение, прочитано из поля или вычислено, ему важен конечный результат. Это и есть инкапсуляция, сокрытие реализации.

3.6.1. Задания для самопроверки

1. Перечислите модификаторы доступа
2. Инкапсуляция -- это
   (a) архивирование проекта
   (b) сокрытие информации о классе
   (c) создание микросервисной архитектуры

3.7. Наследование

3.7.1. Проблема

Второй кит ООП после инкапсуляции -- наследование.

Представим, что есть необходимость создать помимо класса котиков, класс собачек. Данный класс будет выглядеть очень похожим образом, только он будет не мяукать, а гавкать, и заменим обоим животным прыжок на простое перемещение на лапках.

Листинг 8: Класс кота

```
public class Cat {
   private String name;
   private String color;
   private int age;

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

   public String getName() {
     return name;
   }

   public String getColor() {
     return color;
   }

   public int getAge() {
     return age;
   }

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

   public void setColor(String color) {
     this.color = color;
   }

   public void setAge(int age) {
     this.age = age;
   }

   void voice() {
```

```
37        System.out.println(name + " meows");
38    }
39
40    void move() {
41        System.out.println(name + " walks on paws");
42    }
43 }
```

Листинг 9: Класс собаки

```
1  public class Dog {
2     private String name;
3     private String color;
4     private int age;
5
6     public Dog(String name, String color, int age) {
7        this.name = name;
8        this.color = color;
9        this.age = age;
10    }
11
12    public String getName() {
13      return name;
14    }
15
16    public String getColor() {
17      return color;
18    }
19
20    public int getAge() {
21      return age;
22    }
23
24    public void setName(String name) {
25      this.name = name;
26    }
27
28    public void setColor(String color) {
29      this.color = color;
30    }
31
32    public void setAge(int age) {
33      this.age = age;
34    }
35
36    void voice() {
37        System.out.println(name + " barks");
38    }
39
40    void move() {
41        System.out.println(name + " walks on paws");
42    }
43 }
```

Очевидно это не DRY и неприемлемо, если появится необходимость описать классы для целого зоопарка. В приведённых классах есть очень много абсолютно одинаковых и очень похожих полей и методов.

Наследование (англ. inheritance) — концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения.

Наследование в Java реализуется ключевым словом `extends` (англ. -- расширять). И кот и пёс являются животными, у всех описываемых в программе животных есть имя, возраст, окрас, все описываемые животные могут бегать, прыгать, и откликаться на имя. Создав так

называемый **родительский класс**, или суперкласс (листинг 10), и поместив в него поля, геттеры и сеттеры, стало возможным убрать поля, геттеры и сеттеры из кота и пса. Если полей много, лаконичность описания родственных классов может быть весьма ощутимой.

Листинг 10: Класс животного

```
public class Animal {
   private String name;
   private String color;
   private int age;

   public String getName() {
     return name;
   }

   public String getColor() {
     return color;
   }

   public int getAge() {
     return age;
   }

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

   public void setColor(String color) {
     this.color = color;
   }

   public void setAge(int age) {
     this.age = age;
   }

}
```

Чтобы унаследовать один класс от другого, нужно после объявления указать ключевое слово `extends` и написать имя родительского класса как это показано в листингах 11 и 12. Если перенос геттеров возможен, то значит достаточно безболезненно можно перенести и одинаковые методы.

Простой перенос кода в родительский класс показал наличие проблемы. Модификатор `private` определяет область видимости только внутри класса, а если нужно чтобы переменную было видно ещё и в классах-наследниках, нужен хотя бы модификатор доступа по умолчанию. Если же класс наследник создаётся в каком-то другом пакете, то и default не подойдёт.

Листинг 11: Класс собаки

```
public class Dog extends Animal {
   public Dog(String name, String color, int age) {
      this.name = name;
      this.color = color;
      this.age = age;
   }

   void voice() {
      System.out.println(name + " barks");
   }

   void move() {
      System.out.println(name + " walks on paws");
   }
}
```

Листинг 12: Класс кота

```
public class Cat extends Animal {
   public Cat(String name, String color, int age) {
      this.name = name;
      this.color = color;
      this.age = age;
   }

   void voice() {
      System.out.println(name + " meows");
   }

   void move() {
      System.out.println(name + " walks on paws");
   }
}
```

То есть, к членам данных и методам класса можно применять следующие модификаторы доступа

- `private` -- содержимое класса доступно только из методов данного класса;
- `public` -- есть доступ фактически отовсюду;
- default (по-умолчанию) -- содержимое класса доступно из любого места пакета, в котором этот класс находится;
- `protected` (защищенный доступ) содержимое доступно также как с модификатором по-умолчанию, но ещё и для классов-наследников.

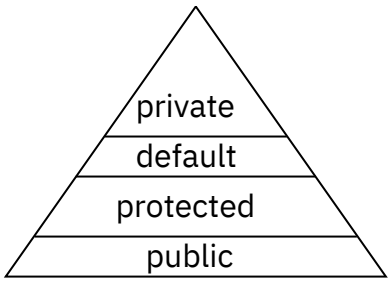


Рис. 21: Модификаторы доступа и их относительная область видимости

То есть верным вариантом в листинге 10 будет применение модификатора `protected`.

3.7.2. Конструкторы в наследовании

> Несмотря на то, что конструктор -- это частный случай метода, если перенести одинаковые конструкторы кота и пса в общий класс животного, программа снова перестанет работать, потому что важно учитывать механику вызова конструкторов при наследовании.

Важно запомнить, что при создании любого объекта в первую очередь вызывается конструктор его базового (родительского) класса, а только потом — конструктор самого класса, объект которого мы создаем. То есть при создании объекта `Cat` сначала отработает конструктор класса `Animal`, а только потом конструктор `Cat`. Но, поскольку конструктор по-умолчанию в нашем случае перестал создаваться, а других может быть бесконечно много, это создало неопределённость, которую программа разрешить не может.

При описании класса, можно явно вызвать конструктор базового класса в конструкторе класса-потомка. Базовый класс еще называют «суперклассом», поэтому в Java для его обозначения используется ключевое слово `super`. Здесь такое же ограничение, как и при вызове конструкторов данного класса (через `this`) -- вызов такого конструктора может

быть только один и быть только первой строкой. Таким образом, код, для всех животных в программе будет выглядеть следующим образом:

Листинг 13: Класс животного

```
public class Animal {
   private String name;
   private String color;
   private int age;

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

   public String getName() {
     return name;
   }

   public String getColor() {
     return color;
   }

   public int getAge() {
     return age;
   }

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

   public void setColor(String color) {
     this.color = color;
   }

   public void setAge(int age) {
     this.age = age;
   }

   void move() {
      System.out.println(name + " walks on paws");
   }
}
```

Листинг 14: Класс кота

```
public class Cat extends Animal {
   public Cat(String name, String color, int age) {
      super(name, color, age);
   }

   void voice() {
      System.out.println(name + " meows");
   }
}
```

Листинг 15: Класс собаки

```
public class Dog extends Animal {
   public Dog(String name, String color, int age) {
      super(name, color, age);
   }

   void voice() {
      System.out.println(name + " barks");
   }
}
```

Листинг 16: Класс птицы

```
public class Bird extends Animal {
   private int flyHeight;

   public Bird(String name, String color, int age,
       int flyHeight) {
      super(name, color, age);
      this.flyHeight = flyHeight;
   }

   void voice() {
      System.out.println(name + " tweets");
   }

   void fly() {
      System.out.println(name + " flies at " +
          flyHeight + " m");
   }
}
```

Для примера, создали ещё один класс, наследника животного, чтобы использовать наследование по назначению. Наследование реализуется через ключевое слово `extends`, расширять. Важно, что класс-родитель расширяется функциональностью или свойствами класса-наследника. Это позволило, например, добавить в птичку такое свойство как высота полёта и такой метод как летать, в дополнение к тому, что умеют все животные.

3.7.3. `Object` и каскадное наследование

Множественное наследование запрещено! Для каждого создаваемого подкласса можно указать только один суперкласс. В Java не поддерживается множественное наследование, то есть наследование одного класса от нескольких суперклассов. Зато возможно каскадное наследование, то есть класс-наследник вполне может быть чьим-то родителем.

Если класс-родитель не указан, таковым считается класс `Object`. Таким образом можно сделать вывод о том, что любой класс в джава так или иначе -- наследник `Object` и, соответственно, всех его свойств и методов. Объект подкласса представляет объект суперкласса, выражаясь проще, возможно ко всем котикам обращаться через общее название Животное, и ко всем объектам в программе возможно обратиться через класс `Object`. Поэтому в программе не будет ошибкой написать подобный код:

```
Object animal = new Animal("Cat", "Black", 3);
Object cat = new Cat("Murka", "Black", 4);
Object dog = new Dog("Bobik", "White", 2);
Animal dogAnimal = new Bird("Chijik", "Grey", 3, 10);
Animal catAnimal = new Cat("Marusya", "Orange", 1);
```

Это так называемое восходящее преобразование (от подкласса внизу к суперклассу вверху иерархии) или **upcasting**. Такое преобразование осуществляется автоматически. Обратное не всегда верно. Например, объект `Animal` не всегда является объектом `Cat` или `Dog`. Поэтому нисходящее преобразование или `downcasting` от суперкласса к подклассу автоматически не выполняется. В этом случае необходимо использовать операцию преобразования типов.

```
Object animal = new Cat("Murka", "Black", 4);
Cat cat = (Cat)animal;
cat.move();
```

Обратите внимание, что в данном случае переменная `animal` приводится к типу `Cat`. И затем через объект `cat` становится возможным обратиться к функционалу кота. Важно при этом, что изначально оператором `new` был создан объект кота, а не `Object` или `Animal`/

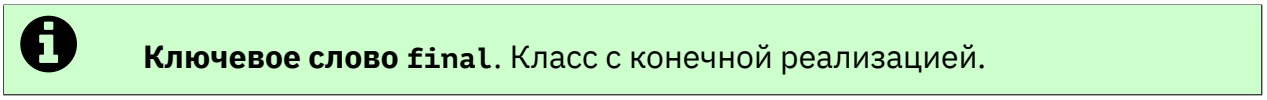
3.7.4. Оператор `instanceof` и ключевое слово `final`

Оператор `instanceof` возвращает истину, если объект принадлежит классу или его суперклассам и ложь в противном случае. Нередко данные приходят извне, и невозможно точно знать, какой именно объект эти данные представляют. Возникает большая вероятность столкнуться с ошибкой преобразования типов. И перед тем, как провести преобразование типов, необходимо проверить возможность выполнить приведение с помощью оператора `instanceof`.

```
Object cat = new Cat("Murka", "Black", 4);
if (cat instanceof Dog) {
   Dog dogIsCat = (Dog) cat;
   dogIsCat.voice();
```

```
} else {
    System.out.println("Conversion is invalid");
}
```

Выражение `cat instanceof Dog` проверяет, является ли переменная `cat` объектом типа `Dog`. Так как в данном случае явно кот не является собакой, то такая проверка вернет значение `false`, и преобразование не сработает. А выражение `cat instanceof Animal` выдало бы истинный результат.



Ключевое слово `final`. Класс с конечной реализацией.

Ключевое слово final может применяться к классам, методам, переменным (в том числе аргументам методов). Применительно к классам, это возможность запретить наследование. То есть, если пометить этим ключевым словом, например, птичку, тогда, если в коде начать писать класс например, попугайчика, и указать, наследование от птицы, то выведется `Cannot inherit from final`.

Рис. 22: Ошибка наследования от final класса

3.7.5. Абстракция

Иногда абстракцию выделяют, как четвёртый принцип ООП.

Абстрактный класс — это написанная максимально широкими мазками, приблизительная «заготовка» для группы будущих классов. Эту заготовку нельзя использовать в чистом виде — слишком «сырая». Но она описывает некое общее состояние и поведение, которым будут обладать будущие классы — наследники абстрактного класса.

Абстрактными могут быть не только классы, но и методы. **Абстрактный метод** -- это метод без реализации. Все животные в примерах выше умеют издавать свой звук. Известно, на этапе проектирования животного, что все животные должны издавать звук, но невозможно сказать, какой именно. Поэтому, определяется, что у животного есть метод издать звук, но реализация этого метода в животном не пишется, слишком мало сведений. Поэтому, метод помечается как абстрактный.

Что будет, если программа попытается вызвать метод `voice()` у животного?

Класс животного максимально абстрактно описывает нужную нам сущность — животное. Но в мире не существует «просто животных». Есть губки, иглокожие, хордовые и т.д. Данный класс теперь слишком абстрактный, чтобы программа могла с ним нормально взаимодействовать, а значит просто является чертежом по которому будут создаваться дальнейшие классы животных. Отметим этот факт явно, написав ключевое слово `abstract` у класса.

— Абстрактный метод -- это метод не содержащий реализации (объявление метода).
— Абстрактный класс -- класс содержащий хотя бы один абстрактный метод.
— Абстрактный класс нельзя инстанциировать (создать экземпляр).

Очевидно, что абстрагирование метода вынуждает абстрагировать класс, но не наоборот, абстрактный класс необязательно должен содержать абстрактные методы, фактически, это просто запрещение создания экземпляров.

3.7.6. Задания для самопроверки

1. Какое ключевое слово используется при наследовании?
   (a) parent
   (b) extends
   (c) как в C++, используется двоеточие
2. super -- это
   (a) ссылка на улучшенный класс
   (b) ссылка на расширенный класс
   (c) ссылка на родительский класс
3. Не наследуются от Object
   (a) строки
   (b) потоки ввода-вывода
   (c) ни то ни другое

3.8. Полиморфизм

Полиморфизм – это возможность объектов с одинаковой спецификацией иметь различную реализацию (Overriding). Полиморфизм выражается возможностью переопределения поведения суперкласса (часто можно встретить утверждение, что при помощи перегрузки. Основная суть в том, что в классе-родителе имеется некоторый метод, но реализация этого метода разная у каждого класса-наследника, фактически это и есть полиморфизм.

Вынесем последний оставшийся в котиках и птичках метод в общий класс животного. В классах-потомках определим такие же методы, как и объявленный метод класса родителя, который хотим изменить.

Листинг 17: Класс кота

```
1 public abstract class Animal {
2    // ...
3
4    void voice();
5
6    // ...
```

```
}

public class Cat extends Animal {
   public Cat(String name, String color, int age) {
      super(name, color, age);
   }

   @Override
   void voice() {
      System.out.println(name + " meows");
   }
}
```

Получается, при наследовании от `Animal`, у которого есть метод `voice()`, класс котика сам определяет, какой он издаёт звук.

Аннотации реализуют вспомогательные интерфейсы.

Аннотация `@Override` проверяет, действительно ли метод переопределяется, а не перегружается. Если существует ошибка в сигнатуре метода, то компилятор сразу об этом скажет.

Полиморфизм чаще всего используется когда нужно описать поведение абстрактного класса или назначить разным наследникам разное поведение, одинаково названное в классе родителе. Но есть и ситуации, когда все классы делают что-то одинаково, а один делает это как-то иначе. Продемонстрируем на примере класса `Snake`, змея.

По очевидным причинам змейка не может ходить на лапках. Поэтому это поведение у змейки будет переопределено.

Листинг 18: Класс кота

```
public class Snake extends Animal {
   public Snake(String name, String color, int age) {
      super(name, color, age);
   }

   @Override
   void move() {
      System.out.println(name + " crawls");
   }

   @Override
   void voice() {
      System.out.println(name + " hisses");
   }
}
```

Также, например, можно было создать черепаху, которая не умеет бегать, рыбу, которая не издаёт звуков, слона, который не умеет прыгать, в отличие от остальных, практикующих «среднее» поведение.

Стоит помнить, что переопределять можно только нестатические методы. Статические методы не наследуются в привычном смысле и, следовательно, не переопределяются. Создав в животном и коте статический метод с одинаковой сигнатурой мы сможем наблюдать то, что называется хайдингом, иначе сокрытием или перекрытием. Также, обратите внима-

ние, что попытка написать у этих методов аннотацию `@Override` вызовет ошибку компиляции.

```
public class Animal {
   // ...
   public static void self() { ... }
}
public class Cat extends Animal {
   // ...
   public static void self() { ... }
}
```

По коду видно, что класс унаследовался и метод должен переопределиться, внешне если создать `Animal a` и `Cat c`, будет похоже, что так и произошло, но если сделать обе переменные типа `Animal`, очевидно, что поведение изменилось. Статические члены класса относятся к классу, т.е. к типу переменной. Поэтому, логично, что если `Cat` имеет тип `Animal`, то и метод будет вызван у `Animal`, а не у `Cat`.

Полиморфизм в языках программирования и теории типов — способность функции обрабатывать данные разных типов. Выделяют параметрический полиморфизм и ad-hoc-полиморфизм.

Широко распространено определение полиморфизма, приписываемое Бьёрну Страуструпу: «один интерфейс — много реализаций». Полиморфизм -- это гораздо более широкое понятие, чем просто переопределение методов, в эту тему завязаны разные интересные теории типов и информации, множество парадигм программирования и другое. С утилитарной точки зрения, остался ещё один вариант, который, тем не менее, не дотягивает до истинного полиморфизма.

К полиморфизму также относится перегрузка методов (Overloading) -- использование более одного метода с одним и тем же именем, но с разными параметрами в одном и том же классе или между суперклассом и подклассами.

Перегрузка работает также, как работала без явной привязки кода к парадигме ООП, ничего нового, но для порядка следует создать возможность животным перемещаться не только абстрактно, но и на какое-то конкретное место или на какое-то конкретное количество шагов.

```
   void move() {
      System.out.println(name + " walks on paws");
   }

   void move(String to) {
      System.out.println(name + " moves to " + to);
   }

   void move(int steps) {
      System.out.println(name + " moves " + steps + " steps away");
   }
```

Как видно, методы имеют одинаковые названия, но отличаются по количеству параметров и их типу.

Чтобы стиль вашей программы соответствовал концепции ООП и принципам ООП в Java следуйте следующим советам:

— выделяйте главные характеристики объекта;
— выделяйте общие свойства и поведение и используйте наследование при создании объектов;
— используйте абстрактные типы для описания объектов;
— старайтесь всегда скрывать методы и поля, относящиеся к внутренней реализации класса.

3.8.1. Задания для самопроверки

1. Является ли перегрузка полиморфизмом
   (a) да, это истинный полиморфизм
   (b) да, это часть истинного полиморфизма
   (c) нет, это не полиморфизм
2. Что обязательно для переопределения?
   (a) полное повторение сигнатуры метода
   (b) полное повторение тела метода
   (c) аннотация Override

Практическое задание

1. Написать класс кота так, чтобы каждому объекту кота присваивался личный порядковый целочисленный номер.
2. Написать классы кота, собаки, птицы, наследники животного. У всех есть три действия: бежать, плыть, прыгать. Действия принимают размер препятствия и возвращают булев результат. Три ограничения: высота прыжка, расстояние, которое животное может пробежать, расстояние, которое животное может проплыть. Следует учесть, что коты не любят воду.
3. * Добавить механизм, создающий 25% разброс значений каждого ограничения для каждого объекта.

*Приложения