***Integrated Development Environment******(IDE)*** – графическая среда интегрированной разработки IDE (Visual Studio, IDEA).

**Горячие клавиши IDEA.**

*Ctrl+Alt+L* – автоматическое форматирование кода.

*Ctrl+Alt+0* – удалить неиспользуемые импорты.

*Alt+Insert* – вызов меню генерации кода.

*Ctrl+Alt+T* – вызов меню генерации тестового класса.

*Ctrl+F* – поиск. *F3* и *Shift* – перемещение по найденным элементам. *Ctrl+R* – замена на этой странице.

*Ctrl+Shift+F* – поиск по указанному пути. *Ctrl+Shift+R* – замена по указанному пути.

*Shift+F6* – переименование переменной.

Двойное нажатие *Shift* – поиск справки.

*Ctrl+Shift+F10* – запуск проекта. *Ctrl+F2* – остановка проекта.

*Ctrl+X* – вырезать текущую строку.

*Ctrl+D* – дублировать текущую строку.

*Ctrl+/* – закомментировать строки как однострочный комментарий.

*Ctrl+Shift+/* – закомментировать строки как многострочный комментарий.

*Ctrl+левый\_клик* или *Ctrl+B* – переход к месту определения метода.

Наведение на метод при нажатом *Ctrl* вызывает подсказку с параметрами метода.

Key Promoter X – плагин напоминает о горячих клавиш при использовании мыши.

Live Templates – плагин запоминает и выводит шаблоны кода.

*sout* – разворачивается в инструкцию *System.out.println();*

*ifn* – разворачивается в конструкцию *if* проверки на равенство *null*.

*fori* – разворачивается в конструкцию *for* с переменное *i* в качестве счетчика.

*iter* – разворачивается в заготовку прохода по списку или массиву.

*System.out.println(); // Вывод сообщения в консоль*

*System.in.read(); // Ввод данных с клавиатуры*

Ввод данных с клавиатуры с фильтрацией ввода:

*import java.util.Scanner; // импорт библиотеки*

*Scanner scanner = new Scanner(System.in); // объявление переменной*

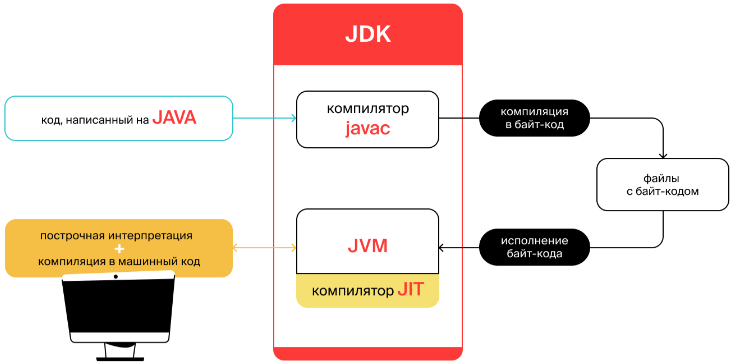
*String command = scanner.nextLine(); // ввод строки из консоли*

С помощью *scanner* можно считать такую информацию:

*nextLine() — строки*

*nextInt() — целые числа*

*nextDouble() — числа с дробной частью*



***Java Virtual Machine (JVM)*** – независимая от платформы исполняющая среда для байт-кода, сгенерированного компилятором (***javac***). Она контролирует выполнение кода и ограничивает его выполнение за пределами исполняющей среды. Эталонной считается *HotSpot JVM* от разработчиков *Java*.

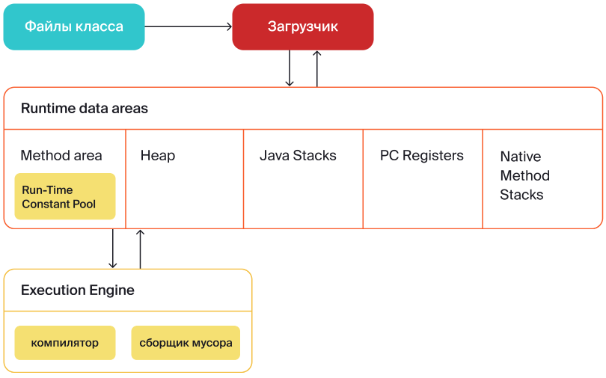
Внутреннее устройство *JVM* можно разбить на три группы:

1. **Подсистема загрузки классов**. Отвечает за поиск и загрузку файлов классов, которые нужны для выполнения программы. Класс ***Platform*** загружает стандартные библиотеки и *JDK*. Класс ***System*** загружает классы самого приложения. Класс ***Bootstrap*** загружает другие загрузчики и пакеты *JRE*.
2. **Подсистема, ответственная за выполнение байт-кода**.

По умолчанию за это отвечает ***JIT***-компилятор (***Just-In-Time***). Он оптимизирует и динамически преобразовывает наиболее часто используемый байт-код в машинный код, и за счет этого повышается производительность приложения.

Альтернативой является статический ***AOT***-компилятор (***Ahead-Of-Time***), который заранее преобразовывает всю программу в машинный код.

1. **Подсистема взаимодействия с памятью**.



***PC Registers*** – локальная область для каждого потока, содержащая адрес инструкции *JVM*, которую поток выполняет в данный момент.

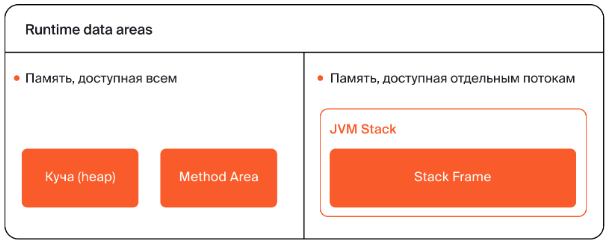
***Java Stacks* (стек)** – локальная область для каждого потока. Она хранит примитивные типы значений и ссылки на объекты кучи. Если потоку требуется больше места в стеке, чем разрешено, то возникает ошибка ***StackOverflowError*** либо ***OutOfMemoryError*** (при динамически расширяемом стеке).

***Heap* (куча)** – это общая область для всех потоков. Она содержит фактические объекты, метаданные классов, массивы и объекты, которые создаются во время работы программы и на которые ссылаются переменные из стека. Создается при запуске *JVM* и уничтожается при завершении её работы. Не освобождается автоматически и управляется ***Garbage Collector*** **(сборщиком мусора)**. Для выделения памяти в куче используется ключевое слово ***new***.

***Method Area* (область метода)** – общая область для всех потоков. Она хранит структуры для каждого класса. Создается при запуске *JVM*.

***Run-Time Constant Pool* (пул констант времени выполнения)** – это структура данных в памяти для каждого класса или типа. Содержит константы, необходимые для запуска кода определенного класса.

***Native Method Stacks* (стеки нативных методов)** – выделяются для каждого потока при его создании. Используются для методов, написанных не на *Java*.



Сборщик мусора работает **на слабой гипотезе о поколениях** – чем младше объект, тем больше вероятность, что он станет не нужным после первого использования. Во время сборки мусора JVM приостанавливается. Если объект является **достижимым** (на него ссылаются), значит его нельзя удалять. Подходы проверки достижимости:

* **Метод подсчета ссылок**. Для каждого объекта в куче создается счетчик ссылок. Сборщик мусора обходит все объекты и оставляет только те, у которых счетчик больше нуля. Из минусов – мусорные объекты могут ссылаться сами на себя; постоянные задержки во время перерасчета счетчика.
* **Метод флагов (или метод маркировки и выметания)**. Для каждого объекта в памяти хранится бит, указывающий на его достижимость. Изначально все объекты помечаются как недостижимые. Не проверяются корневые объекты.
* **Метод карточного стола**. Представляет объекты старшего поколения в виде набора карт. Если старший объект ссылается на молодой, то карта переворачивается и при следующей проверке на достижимость проверяются только объекты, доступные “из-под этой карты”. Сборщик мусора может убрать только часть кучи.

Область в куче делится на область младшего (***Eden, S0, S1***) и область старшего (***Tenured***) поколений. Это позволяет избавиться от фрагментации памяти. Если места не хватает в области старшего поколения, то запускается полная сборка мусора в обоих областях.

*HotSpot JVM* включает в себя три типа сборщиков с разными характеристиками производительности:

* ***Serial* (последовательные)** – используют один поток для сборки мусора. Подходит для однопроцессорных пк и объемом данных до 100Мб. Требует полную остановку приложения.
* ***Parallel* (параллельные)** – используют несколько потоков для сборки мусора. Подходят для многопроцессорных пк и средним объемом данных. Требует полную остановку приложения.
* ***Concurent* (конкурентные)** – многопоточные сборщики мусора. Большая часть работ выполняется, не останавливая потоки приложения, а конкурирую с ними. В *Java* 11 по умолчанию используется сборщик мусора ***G1***. Начиная с *Java* 15, альтернативой является сборщик мусора ***ZGC***.

**Основные характеристики *JVM*:**

* пропускная способность определяет пиковую нагрузку
* время отклика определяет как приложение справляется с остановками на сбор мусора
* размер используемой памяти, необходимый для эффективной работы сборщика мусора

**Основные принципы настройки сборщика мусора:**

* Максимальное число объектов должно быть удалено во время малой сборки. Это позволит избежать полной остановки приложения.
* Чем больше памяти выделено, тем лучше работает сборка мусора.
* Для эффективной работы приложения нужно настроить две из трех основных характеристик JVM.

Задать эти настройки можно с помощью флагов при запуске *JVM*.

Java чувствителен к регистру имен переменных и методов. В именах допускается использовать цифры (но нельзя начинать имя с цифры), \_ и $. В Java не поддерживаются целочисленные значения без знака. Однострочные комментарии обозначаются //, многострочные - /\* \*/.

**Внутренние типы данных**

|  |  |  |
| --- | --- | --- |
| **Обозначение** | **Диапазон** | **Описание** |
| bool | true или false | Признак истинности или ложности |
| byte | от -128 до 127 | 8-битное число |
| short | от -32 768 до 32 767 | 16-битное число |
| int | от -2 147 483 648 до 2 147 483 647 | 32-битное число |
| long | от -9 223 372 036 854 775 808  до 9 223 372 036 854 775 807 | 64-битное число |
| char | от U+0000 до U+ffff | Одиночный 16-битный символ Unicode |
| float | от -3.4х10^38 до 3.4х10^38 | 32-битное число с плавающей точкой |
| double | от +-5.0х10^(-324) до +-1.7х10^308 | 64-битное число с плавающей точкой |

При записи чисел типа *long* необходимо добавлять букву *L* в конце. Переменную *char* можно инкрементировать, а также присваивать целочисленные значения. Данный тип задается парными одинарными кавычками (например, ‘k’). Тип *float* хранит 38 знаков после запятой, а тип *double* - больше 300 знаков после запятой. В коде вместо 10 ставится буква *Е*. Все примитивы хранят конкретное значение.

**Управляющие последовательности**

|  |  |  |  |
| --- | --- | --- | --- |
| \’ | Одинарная кавычка | \f | Перевод страницы |
| \” | Двойная кавычка | \t | Горизонтальная табуляция |
| \\ | Обратная косая черта | \b | Возврат на одну позицию |
| \r | Возврат каретки | \ddd | Восьмеричная константа |
| \n | Перевод строки | \uxxxx | Шестнадцатеричная константа |

Не рекомендуется использовать синтаксис множественных объявлений или инициализации переменных, т.к. это мешает восприятию и сбивает с толку.

Префиксная форма (++x) записи операций инкремента и декремента изменяет значение до ей использования, а постфиксная (x++) – после.

**Логические операторы**

|  |  |  |  |
| --- | --- | --- | --- |
| & | И (побитовое) | && | Укороченное И |
| | | ИЛИ (побитовое) | || | Укороченное ИЛИ |
| ^ | Исключающее ИЛИ | ! | НЕ |

Побитовые операторы неактуальны.

Укороченные операторы могут не вычислять второй оператор, если результат уже не изменится.

По приоритетам – отрицание (!), операции в скобках, логическое И (&&), логическое ИЛИ (||).

Допускается **неявное преобразование типов**. Например, byte в int.

Для **явного преобразования типов** нужно явно указать целевой тип. Например, здесь потеряется дробная часть в результате преобразования:

*double x = 10.0, y = 3.0;*

*int i = (int) (x / y);*

В *java* реализованы условные конструкции ***if-else*** и ***if-else-if***.

**Условная операция** **(тернарный оператор)** (?:) является сокращенным способом написания простого оператора if-else.

*условие ? первое\_выражение : второе\_выражение;*

Например:

*absval = val < 0 ? -val : val;*

*или*

*if(val < 0) absval = -val;*

*else absval = val;*

Также реализована условная конструкция ***switch***. Условное выражение может быть типами *int*, *char, byte, short* или *String*. При отсутствии ключевого слова *break* сначала выполняются инструкции текущей ветви, а затем следующей за ней и т.д. Поддерживаются вложенные условные конструкции *switch*.

*switch (month) {*

*case 1: // январь*

*case 3: // март*

*case 5: // май*

*case 7: // июль*

*case 8: // август*

*case 10: // октябрь*

*case 12: // декабрь*

*System.out.println("В этом месяце 31 день.");*

*break;*

*case 2: // февраль*

*System.out.println("В этом месяце 28 или 29 дней.");*

*break;*

*case 4: // апрель*

*case 6: // июнь*

*case 9: // сентябрь*

*case 11: // ноябрь*

*System.out.println("В этом месяце 30 дней.");*

*break;*

*default:*

*System.out.println("Месяца с таким номером не существует."); }*

Цикл ***for*** подходит для тех случаев, когда заранее известно количество итераций. Цикл *for* не ограничивается определением и итерацией только одной переменной, их может быть несколько. В условии цикла не обязательно должна находиться переменная его итерирования. Цикл *for* позволяет при объявлении пропускать одно, два или все условия.



Цикл ***while*** используется для расчётов, когда работа программы зависит от внешних данных, которые изначально неизвестны — например, их вводит пользователь.

*while (условие) {*

*последовательность инструкций*

*}*

Цикл ***do-while*** используется для выполнения инструкций до проверки условия цикла.

*do {*

*последовательность инструкций*

*} while (условие);*

Для выхода из любого цикла можно использовать ключевое слово ***break***. Для досрочного завершения текущей итерации и переходу к следующей используется ключевое слово ***continue***.

В коде со множеством вложенных циклов можно использовать переход по меткам. **Метка** – это имя, обозначающее блок кода. Например:

*for(i=l; i<4; i++) {*

*one: {*

*two: {*

*three: {*

*System.out.println("\ni равно " + i);*

*if(i==l) break one; // Переход по метке*

*if(i==2) break two;*

*if(i==3) break three; }}}}*

**Массив** – это набор однотипных элементов данных, для доступа к которым применяется числовой индекс. Индексация массивов начинается с нуля. Если массив объявлен, но его элементы не заданы явным образом, то они получат стандартные значения в соответствии с их типом. Массивы являются ссылочными типами.

**Индекс** — номер элемента в массиве, его тип значения всегда *int*. Узнать длину массива можно с помощью вызова ***length***.

Объявление массива:

*String[] currencies = {"USD", "EUR"};*

или

*String[] currencies = new String[2];*

*currencies[0] = "USD";*

*currencies[1] = "EUR";*

Пример объявления **двухмерного массива**:

*int table[][] = new int[2][3];*

*table[0][0] = 1;*

*table[0][1] =2;*

*…*

или

*int table[][] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} }*

Перебор массива циклом *for*:

*for (int i = 0; i < currencies.length; i++) {*

*System.out.println("Поддерживаемая валюта: " + currencies[i]); }*

Перебор массива циклом *for-each*:

*for (String s : currencies) {*

*System.out.println("Поддерживаемая валюта: " + s); }*

Другие многомерные массивы объявляются и инициализируются аналогичным образом.

**Нерегулярные массивы** позволяют выделить память под массив на основании первого (самого левого) размера. Это позволяет определить размер второго измерения вручную (они могут различаться по размеру):

*int table[][] = new int[3][];*

*table[0] = new int[10];*

*table[1] = new int[10];*

*table[2] = new int[2];*



**Перечисления** – это список именованных констант, определяющих новый тип данных. Объекты перечислений могут иметь значения только из этого списка. Для объявления перечислений используется ключевое слово ***enum***. Константы являются неявно ***static final*** членами перечислений.

Для сравнения констант перечислимого типа используется оператор сравнения (==). Перечисления можно использовать в качестве селектора в переключателе *switch*.

*enum Transport {CAR, AIPLANE, TRAIN}*

*class EnumDemo {*

*public static void main(String args[]) {*

*Transport tp = Transport.AIPLANE; // Инициализация без new*

*if (tp == Transport.TRAIN) // Можно сравнивать через ==*

*System.out.println(“tp содержит TRAIN\n”);*

*switch (tp) {*

*case CAR: break;*

*case AIPLANE: break;*

*case TRAIN: break; } } }*

По умолчанию все перечисления реализуют два метода:

*public static перечислимый\_тип[] values()*

*public static перечислимый\_тип valueOf(String str)*

Первый возвращает массив со списком констант перечисления, а второй – константу перечисления, которая соответствует строке *str*.

*Transport allTransport[] = Transport.values();*

*tp = Transport.valueOf(“AIRPLANE”);*

В Java перечисления реализованы как типы классов, но для их объявления **не требуется слово** ***new***. Допускается объявление конструкторов, методов и переменных, а также реализация интерфейсов. В этом случае после объявления констант ставится точка с запятой.

Перечисления не могут быть подклассами и суперклассами, однако они наследуют переменные и методы класса ***java.lang.Enum***. Основными методами являются ***ordinal()*** и ***compareTo()***. Первый возвращает порядковое значение константы (начиная с нуля), а второй сравнивает эти порядковые значения.

*if (tp.compareTo(tp2) < 0)*

*System.out.println(tp + “ идет перед ” + tp2);*

К *enum* можно применять методы *ToString(), getClass(), hashCode()* и другие.

**Строка (*String*)**,начиная с *java* 9, является массивом байтов (выделяет на символ столько места, сколько требуется). Значения ***String*** являются константными (строковые литералы), поэтому они не меняются, а создаются новые строки.

Все строки являются ссылочными типами. Строки можно объединять в массивы. При создании строк с помощью литерала (не через *new*) они попадают в особое хранилище – пул строк. При повторном создании идентичной строки вернется ссылка на уже существующую строку.

Метод ***equals(String)*** используется для сравнения строк и возвращает булево значение.

Метод ***toString()*** возвращает строковое представление объекта, а при вызове для строк просто возвращает их самих.

Метод ***length()*** возвращает количество символов в строке.

Метод ***isEmpty()*** возвращает *true*, если строка является пустой.

Метод ***isBlank()*** возвращает *true*, если строка пустая или содержит только пробельные символы.

Метод ***trim()*** возвращает строку, из которой удалены пробельные символы в начале и в конце.

Методы ***toUpperCase()*** и ***toLowerCase()*** переводят строки в верхний и нижний регистры.

Метод ***split(String regex)*** превращает строку в массив строк по разделителю и возвращает его. Для использования спец.символов в качестве разделителя, их нужно экранировать с помощью **\\** .

Метод ***join(String delimeter, String … strs)*** превращает массив подстрок в строку с разделителем и возвращает её. Запись ***String … strs*** называется ***variable arguments (varargs)*** и обозначает, что метод может принять любое количество аргументов или массив.

**Подстрока** **(*substring*)** – это непрерывный набор символов внутри строки. Символы в подстроках нумеруются как в массиве (с нуля).

Метод ***charAt(int)*** возвращает символ, расположенный по указанному индексу.

Методы ***indexOf(String)*** и ***lastIndexOf(String)*** возвращают индекс начала подстроки (отличаются направлением поиска). Если подстрока не найдена, вернется -1. Для поиска с конкретного символа используется перегрузка этих методов - ***indexOf(String, int)*** и ***lastIndexOf(String, int)***.

Метод ***contains(String)*** возвращает *true*, если подстрока будет найдена в любом месте строки.

Метод ***startsWith(String)*** возвращает *true*, если подстрока будет найдена в начале строки.

Метод ***endsWith(String)*** возвращает *true*, если подстрока будет найдена в конце строки.

Метод ***substring(int)*** возвращает подстроку, начиная с указанного индекса и до конца строки.

Метод ***substring(int firstIndex, int lastIndex)*** возвращает подстроку, указанную между индексами, включая первый и не включая последний индексы.

Метод ***replace(String target, String replacement)*** возвращает новую строку, в которой заменены все вхождения подстроки *target* на подстроку *replacement*.

Метод ***replaceFirst(String target, String replacement)*** возвращает новую строку, в которой заменено первое вхождение подстроки *target* на подстроку *replacement*.

Для представления строки в виде набора редактируемых символом используется класс ***StringBuilder***. У него есть изменяемый размер буфера, как у списка. Вместимость можно задать при создании экземпляра через конструктор, принимающий *int*.

Метод ***toString()*** преобразует содержимое *StringBuilder* в неизменяемую строку и возвращает её.

Класс *StringBuilder* поддерживает методы *indexOf(String)*, *lastIndexOf(String)*, *substring(int)* и *substring(int beginIndex, int endIndex)*.

Метод ***append(String)*** добавляет подстроку в конец.

Метод ***insert(int index, String str)*** добавляет подстроку в середину (со сдвигом вправо).

Метод ***replace(int indexFromInclusive, int indexToExclusive, String replacement)*** заменяет обозначенный промежуток на указанную подстроку. Первый индекс берется включительно, а второй - нет.

Методы ***deleteCharAt(int)*** и ***delete(int startIndexInclusive, int endIndexExclusive)*** удаляют элементы по указанным индексам. Первый индекс берется включительно, а второй - нет.

Метод ***reverse()*** позволяет отзеркалить (перевернуть) строку.

Метод ***setLength(int)*** обрезает строку до указанного размера.

**Палиндром** – это число, сочетание букв или текста, которые читаются одинаково как слева направо, так и справа налево.

Для форматирования вывода используется метод ***String.format(String format, Object… args)***:

*String[] colors = new String[]{"красный", "жёлтый", "зелёный"};*

*String trafficLight = String.format("Цвета светофора: %s, %s и %s.", colors[0], colors[1], colors[2]);*

**Популярные символы преобразования**

|  |  |  |  |
| --- | --- | --- | --- |
| Обозначение | Тип | Обозначение | Тип |
| %s | String | %d | Целое число |
| %S | Вывод строки в верхнем регистре | %c | char |
| %b | boolean | %t | Date |
| %B | boolean в верхнем регистре | %% | символ % |
| %f | Вещественное число с плавающей точкой | %n | Перенос строки в форм.строках |

Также для форматирования можно использовать метод ***System.out.printf((String format, Object… args)***.

*System.out.printf("%b", "Панама") // выведет true (преобразование строк)*

*System.out.printf("%25s", pizza); // минимальное число символов в строке = 25 (по правому краю)*

*System.out.printf("%-25s", pizza); // теперь выровняем текст по левому краю*

*System.out.printf("%1.12s", pizza); // в строке теперь от 1 до 12 символов*

*System.out.printf("%2.2f", price); // минимум 2 символа и ровно 2 символа после запятой*

***SOLID*** – это набор принципов, следуя которым, программный код будет более чистым и гибким.

***S****ingle-responsibility* (единая ответственность) – один класс или файл должен иметь только одну цель и одно единственное назначение.

***O****pen-closed* (принцип открытости/закрытости, или наследования) – программные сущности должны быть открыты для расширения, но закрыты для модификации.

***L****istov substitution principle* (принцип подстановки Барбары Лисков) – поведение методов в дочернем классе должно следовать принципам базового класса, а не изменять его. Т.е. если базовый метод возвращал высоту, то и дочерний должен возвращать высоту, а не длину/скорость или что-то иное.

***I****nterface segregation* (принцип разделения интерфейса) – лучше создавать много отдельных интерфейсов, чем один, который включает в себя все функции.

***D****ependency invercion* (инверсия зависимостей) – модули верхних уровней не должны зависеть от модулей нижних уровней. Т.е. пишем код так, чтобы он имел как можно меньше зависимостей.

Одним из главных свойств Java является поддержка **объектно-ориентированного программирования** (ООП). В языках ООП определяются данные и процедуры, которым разрешается обрабатывать эти данные. Таким образом, тип данных определяет те операции, которые применимы к этим данным. Во всех языках ООП поддерживаются три основных принципа ООП: инкапсуляция, полиморфизм и наследование.

1. **Инкапсуляция** характерная черта, описывающая способность языка скрывать лишние детали реализации от пользователя объекта и предохраняющая целостность данных (черный ящик).

Данные и код внутри объекта могут быть закрытыми (**private**) или открытыми (**public**). Закрытый код или данные доступны только элементам, содержащимся в том же самом объекте. Если код или данные являются открытыми, то к ним можно обращаться из любой части программы. Как правило, открытые элементы объекта используются для создания управляемого интерфейса к его закрытым элементам. Основной языковой конструкцией, поддерживающей инкапсуляцию, является класс.

1. **Полиморфизм** – способность языка трактовать связанные объекты в сходной манере. Она позволяет реализовывать одно и то же действие в зависимости от контекста (например, рисовать разные геометрические фигуры).
2. **Наследование** – процесс, в ходе которого один объект приобретает свойства другого объекта. Это отражает возможность языка разрешать построение определений новых классов на основе определений существующих классов (стимулирование многократного использования уже написанного кода).

**Класс** – это определяемый пользователем тип, состоящий из полей данных (переменных-членов) и членов, которые оперируют полями данных (конструкторы, свойства, методы, события и т.д.). **Объектами** называются экземпляры классов в памяти. Для обращения к переменных-членам класса используется **точечная нотация**. Классы являются ссылочными типами.

Создание класса, как типа данных:

*class имя\_класса {*

*// описание переменных-членов }*

Создание объекта класса:

*имя\_класса имя\_объекта\_класса // объявление ссылки на объект*

*имя\_объекта\_класса = new имя\_класса(список параметров); // выделение памяти для объекта*

или сразу

*имя\_класса имя\_объекта\_класса = new имя\_класса(список параметров);*

**Конструктор** – это специальный метод класса, который неявно вызывается при создании объекта с применением слова ***new***. Он позволяет установить состояние объекта в момент его создания. Конструктор не имеет возвращаемого значения и всегда именуется идентично имени класса, объекты которого он конструирует.

**Конструктор по умолчанию** есть у всех классов, он инициирует все переменные экземпляра их значениями по умолчанию (ссылочные типы – null, логические – false, остальные – нулями).

**Простой конструктор** позволяет задавать нужные стандартные значения по умолчанию. Например:

*class MyClass {*

*int x;*

*MyClass() {*

*x = 10; } }*

**Cпециальный конструктор** работает аналогично перегрузке метода (с тем же самым именем, но разным количеством или типами аргументов).

*class MyClass {*

*int x;*

*MyClass(int i) {*

*x = i; } }*

При наличии неоднозначности в именах для доступа класса к собственным данным или членам используется слово **this**. Например:

*class Motorcycle {*

*public String driverName;*

*public void SetDriverName (String driverName) => this.driverName = driverName; }*

Конструкторы можно перегружать.

Через ***this*** можно обратиться к другим конструкторам, чтобы не дублировать код:

*class Bot {*

*String name;*

*public Bot() { // конструктор без параметров*

*System.out.println("Чат-бот создан"); }*

*public Bot(String name) { // конструктор с параметром*

*this(); // вызвали конструктор без параметров*

*this.name = name;*

*System.out.println("Вас приветствует " + name); } }*

**Методы** – это подпрограммы, которые манипулируют данными. Во многих случаях они предоставляют доступ к закрытым данным класса. Имя метода лучше начинать с глагола.

*возвращаемый\_тип имя\_метода(список параметров) {*

*// тело метода }*

Ключевое слово ***return*** позволяет завершить текущий метод и вывести возвращаемое значение заданного типа (при наличии).

Значения аргументов могут передаваться методу двумя способами:

1. **Вызов по значению**. В этом случае метод копирует значение аргумента и не может изменить исходный аргумент. Например:

*void noChange(int i, int j) {*

*i = i + j;*

*j = -j; }*

1. **Вызов по ссылке**. В этом случае метод получает ссылку на аргумент и оказывает на него влияние.

*int a, b;*

*Test (int i, int j) {*

*a = i;*

*b = j; }*

*void change(Test ob) {*

*ob.a = ob.a + ob.b;*

*ob.b = -ob.b; }*

**Перегрузка методов** – набор идентично именованных методов, которые отличаются друг от друга количеством или типами параметров.

Ключевое слово ***static*** используется для обозначения глобальных членов класса, которые можно использовать на уровне класса без ссылки на объект (принадлежит классу, а не объекту). Они будут общими для всех объектов класса. Например, метод *sqrt()* класса *Math*. Ограничения статичных методов:

* Допускается использовать только статических переменных, определенных в его классе.
* Допускается вызов только других статических методов.
* Должна отсутствовать ссылка *this* и *super*.

**Статические блоки** (аналоги статических конструкторов) – это набор инструкций, который выполняется до первого использования класса. Например, в нем можно инициализировать начальные значения статичных переменных:

*class StaticBlock{*

*static double rootOf2;*

*static double rootOf3;*

*static{ rootOf2 = Math.sqrt(2.0);*

*rootOf3 = Math.sqrt(3.0); }*

Статические переменные и методы нельзя перезаписать, но можно переделать через сокрытие.

**Вложенный класс** — это такой класс, который объявляется в другом классе. Вложенные классы, объявленные без применения ключевого слова *static,* являются **внутренними**. Внутренний класс имеет доступ ко всем элементам внешнего класса.

Для передачи списка параметров не фиксированной длины используется **метод с переменным числом аргументов**. Такой список должен передаваться последним аргументом. Методы с переменным числом аргументов также можно перегружать, но лучше отказаться от этого в связи с возможной неоднозначностью при вызовах метода. Например, следующий код объявляет аргументами строку *msg* и массив *v* типа *int* :

*static void vaTest(String msg, int … v) { }*

Наследуемый класс называется **суперклассом**, а наследующий – **подклассом**. Наследование объявляется с помощью ключевого слова ***extends***:

*class имя\_подкласса extends имя\_суперкласса*

*class Circle extends TwoDShape*

Множественное наследование не поддерживается в Java – у каждого подкласса может быть только один суперкласс.

Подклассы не имеют доступ к закрытым переменным суперкласса. Для доступа к закрытым переменным и сохранения инкапсуляции класса, рекомендовано создать и использовать геттеры и сеттеры – специальные методы ***get*** и ***set***. Например:

*double getHeight() { return height; }*

*void setHeight(double h) { height = h; }*

**Сокрытие** – объявления поля в подклассе с таким же именем, что и поле в суперклассе.

Объект наследника можно сохранить в переменной родительского типа.

Если конструктор определен только в подклассе, то конструируется объект подкласса, а родительская часть создается конструктором по умолчанию (без параметров).

Если конструктор определен в подклассе и в суперклассе, то выполняются оба конструктора. Для вызова конструктора суперкласса используется ключевое слово ***super*** – будет вызван конструктор ближайшего суперкласса, который соответствует указанным аргументам. Это справедливо и для многоуровневой иерархии.

*class TwoDShape {*

*private double width;*

*private double height;*

*TwoDShape(double w, double h) {*

*width = w;*

*height = h; }*

*class Triangle extends TwoDShape {*

*private String style;*

*Triangle(String s, double w, double h) {*

*super(w, h);*

*style = s; }*

При создании объекта конструкторы вызываются в порядке наследования, начиная с суперкласса и заканчивая подклассом. Метод ***super()*** должен быть первым в инструкции конструктора подкласса. Если в классе-родителе есть конструктор без параметров, то в конструкторе подкласса компилятор вызовет его автоматически, неявно.

Допускается конструирование объекта на основе другого объекта:

*TwoDShape(TwoDShape ob) {*

*width = ob.width;*

*height = ob.height; }*

Ключевое слово ***super*** также используется для обращения к переменным и методам суперкласса, скрываемых переменными и методами подкласса (одно и то же имя). Например:

*B(int a, int b) {*

*super.i = a; // переменная i из класса A*

*i = b; } // переменная i из класса B*

Вызвать родительский метод можно также через точечную нотацию *super.someMethod()*.

**Аннотация** позволяет отслеживать изменения суперкласса и сигнализировать об этом при компиляции.

Полиморфизм предоставляет подклассу способ определения собственной версии метода, определенного в его базовом классе, с применением процесса, который называется **переопределением метода**. Переопределение (или **перегрузка**) метода происходит, когда сигнатуры (имя и аргументы) переопределяемого и переопределяющего методов совпадают. Например, переопределим метод *area()*:

*class TwoDShape {*

*double area() {*

*return 0.0 } }*

*class Triangle extends TwoDShape {*

*@Override // аннотация переопределения метода (необязательна)*

*double area() {*

*return getWidth() \* getHeight() / 2 ; } }*

Специальный класс ***Object*** является суперклассом для всех остальных классов. Он всегда неявно наследуется. Ссылочная переменная типа *Object* может ссылаться на объект любого класса.

Для сравнения объектов собственных классов и корректного поиска по спискам нужно **переопределить** метод ***equals(Object)***. Для корректного поиска по хэш-таблицам нужно **переопределить** методы ***equals(Object)*** и ***hashCode()***, т.к. одинаковые объекты должны иметь одинаковый хеш.

*import java.util.Objects;*

*import java.util.Arrays;*

*import java.util.Objects;*

*@Override*

*public boolean equals(Object o) {*

*if (this == o) return true; // проверяем адреса объектов*

*if (o == null || getClass() != o.getClass()) return false; // проверяем ссылку на null и сравниваем классы*

*Book otherBook = (Book) o; // открываем доступ к полям другого объекта*

*return Objects.equals(title, otherBook.title) && // проверяем все поля*

*Objects.equals(author, otherBook.author) && // нужно логическое «и»*

*(pagesNumber == otherBook.pagesNumber); } // примитивы сравниваем через ==*

*@Override*

*public int hashCode() {*

*// вызываем вспомогательный метод и передаём в него нужные поля*

*return Objects.hash(title, author, pagesNumber); }*

Для получения текущего состояния объекта в виде текста используется метод ***toString()***. В связи с тем, что его базовая реализация неинформативна, рекомендуется его всегда **переопределять**.

*class Address {*

*public String city;*

*public String street;*

*public int houseNumber;*

*@Override // переопределяем toString*

*public String toString() {*

*return city + ", " + street + ", д. " + houseNumber; } } // вызов println(address) вернет эту строку*

Метод ***Arrays.toString(Object[] o)*** проверяет массив на *null* и возвращает его текстовое представление.

Классы и методы имеют модификаторы доступов – открытые (***public***), закрытые (***private***) и защищенные (***protected***).

Закрытые члены доступны только внутри класса, в котором определены.

Защищенные члены могут использоваться классом, который их определяет, и любым дочерним классом. Однако, они недоступны внешнему миру.

Если модификатор не указан, то используется модификатор по умолчанию – ***default*** или ***package-private***.

**Пакет** – область в **пространстве имен**, объединяющая логически связанные классы. В одном пространстве имен все классы должны обладать уникальным именем.

В пакетах определены специальные права доступа к содержимому пакета. Это обеспечивает дополнительный уровень инкапсуляции.

Чтобы создать пакет, достаточно поместить инструкцию ***package*** в начало файла. В результате классы, определенные в этом файле, будут принадлежать указанному пакету. Поскольку пакет определяет пространство имен, имена классов, содержащиеся в файле, войдут в это пространство имен как его составные части. Класс не может относиться к нескольким пакетам сразу.

*package имя\_пакета;*

**Уровни доступа к членам классов**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Закрытый член | Член, доступный по умолчанию | Защищенный член | Открытый член |
| Доступен в том же классе | Да | Да | Да | Да |
| Доступен из подкласса в том же пакете | Нет | Да | Да | Да |
| Доступен из любого класса в том же пакете | Нет | Да | Да | Да |
| Доступен из подкласса в любом пакете | Нет | Нет | Да | Да |
| Доступен из всякого класса в любом пакете | Нет | Нет | Нет | Да |

Исходя из таблицы, чтобы класс и его методы стали доступны за пределами пакета, в котором он определен, необходимо объявить открытыми (*public*) класс, его конструктор и методы.

Для того чтобы воспользоваться классом в другом пакете, нужно применить инструкцию ***import*** либо указать полное имя класса, предваренное именем пакета. Чтобы импортировать все содержимое пакета, вместо имени класса следует указать звездочку (**\***). Инструкция *import* должна следовать сразу за инструкцией *package*. Пакеты могут быть вложенными (пакет с пакетами).

*import имя\_пакета.имя\_класса;*

*import имя\_пакета.\*;*

Популярные пакеты – *java.lang* (импортируется автоматически), *java.io* , *java.net* , *java.applet* , *java.awt* .

В java поддерживается механизм **статического импорта**, позволяющего ссылаться на статические члены по их простым именам без дополнительного указания имени класса. Например, класс *Math*:

*import static java.lang.Math.\*; // для всех статических членов*

*import static java.lang.Math.sqrt; // для одного статического члена*

*double x = sqrt(2);*

**Абстрактный метод** – это член базового класса, который не представляет базовую реализацию, а предлагает только сигнатуру.

**Абстрактный класс** – это базовый класс, объекты которого нельзя создать. При его создании используется ключевое слово **abstract**. Если в классе объявлен абстрактный метод, то он обязан тоже быть абстрактным. Подкласс наследующий абстрактный класс, должен реализовывать все методы суперкласса. В абстрактном классе могут быть определены конкретные методы без указания ключевого слова *abstract*.

В абстрактный класс выносятся общие для всех наследников поля и методы.

Например, просто объектов *TwoDShape* не существует, он должен быть либо *Triangle*, либо *Circle*. Идея фигуры является допустимой, но это не конкретная сущность.

*abstract class TwoDShape {*

*abstract double area(); }*

*class Triangle extends TwoDShape {*

*@Override*

*double area() {*

*return getWidth() \* getHeight() / 2 ; } }*

Конструктор наследника неявно использует конструктор абстрактного родителя.

Методы и классы, объявленные с использованием ключевого слова ***final***, не могут быть переопределены и унаследованы.

Ключевое слово ***final*** также используется для объявления именованных констант переменных.

Для объявления константы класса используются оба слова ***static*** и ***final*** в любой последовательности:

*static final тип ИМЯ\_КОНСТАНТЫ = значение;*

**Интерфейс** – это коллекция абстрактных членов (методов), не представляющих свои стандартные реализации. Он добавляет нескольким классам определенный функционал. Классы обязаны иметь реализацию всех методов интерфейсов, которые они поддерживают. Интерфейсы объявляются с помощью ключевого слова ***interface*** и могут быть реализованы в любом количестве классов.

*уровень\_доступа interface имя {*

*возвращаемый\_тип имя\_метода\_1 (список\_параметров);*

*тип переменная\_1 = значение; }*

Уровень доступа может быть *public* (доступен в любом пакете) либо не указывается (доступен только внутри пакета). Методы, объявленные в интерфейсе, по умолчанию неявно считаются *public*. Переменные интерфейса неявно обозначаются ключевыми словами *public*, *final*, *static* и являются константами, которые обязательно подлежат инициализации при объявлении интерфейса.

Для реализации интерфейса в классе используется ключевое слово ***implements***:

*class имя\_класса extends суперкласс implements интерфейс\_1, интерфейс\_2 {*

*// тело класса }*

Интерфейсы поддерживают наследование с помощью ключевого слова ***extends***. Если класс реализует один интерфейс, наследующий другой, то в нем следует определить все методы, объявленные в интерфейсах по всей цепочке наследования.

Поля переменных в интерфейсе всегда являются константами.

Нельзя создать объекты на основе интерфейса, но можно объявить переменную с типом интерфейса и присвоить ей в качестве значения любой объект, который реализует данный интерфейс. Грубо говоря, мы берем объект и обрезаем все лишнее, чтобы он поместился в родительский класс.

*MailClient drone = new Drone(); // Интерфейс переменная = новый объект*

Начиная с JDK 8, интерфейсы могут иметь **реализацию по умолчанию** (они также не могут иметь экземпляров объектов). Они используются, если другие реализации не предоставляются. Это обеспечивает сохранение работоспособности существующего кода, даже если интерфейс был обновлен, и не реализовывать опциональные методы интерфейса. Для реализации интерфейсов по умолчанию используется ключевое слово ***default***. При желании можно перегрузить метод интерфейса по умолчанию.

*public interface MyIF {*

*int getUserID();*

*default int getAdminID() {*

*return 1; } }*

Реализация методов, определенная в классе, всегда имеет более высокий приоритет по сравнению с методами по умолчанию.

Если класс реализует два интерфейса с одинаковыми именами методов по умолчанию и не переопределяет их, то возникнет ошибка.

Если класс реализует два интерфейса с одинаковыми именами методов по умолчанию, не переопределяет их и один из этих интерфейсов наследует другой, то будет применяться реализация метода более расширенного интерфейса.

На заданную по умолчанию реализацию можно ссылаться явно с помощью ключевого слова ***super***:

*Alpha.super.reset();*

Начиная с JDK 8 интерфейс может определять статические методы. Их также можно вызывать только без экземпляров объектов. Для вызова такого метода в классе, реализовывающем такой интерфейс, нужно использовать следующую форму:

*имя\_интерфейса.имя\_статического\_метода();*

Начиная с JDK 9 интерфейс может включать закрытые методы, которые вызываются только методом по умолчанию или другими закрытыми методами. Они позволяют нескольким методам по умолчанию использовать один и тот же фрагмент кода, не дублируя его.

**Полиморфный класс** – класс, который связан отношением ***is-a*** с более чем одним типом (*Object*).

**Классический полиморфизм** – способность языка одинаково работать с объектами (поддержка наследуемых методов), если они имеют одинаковый интерфейс или родительский класс.

**Динамический *ad-hoc* полиморфизм** – заключается в переопределении методов. Он позволяет им демонстрировать различное поведение при вызове для разных типов.

**Статический *ad-hoc* полиморфизм** – заключается в перегрузке методов. В классе может быть несколько методов с одинаковым названием, но с разными типами параметров.

**Параметрический полиморфизм** – позволяет реализовать поведение класса или метода с указанием параметров и возвращаемых значений общего вида вместо конкретных (не зависимо от типа данных).

Автоматическое приведение ссылочных типов происходит при присваивании объекта класса-наследника переменной с типом интерфейса или базового класса.

*Document document = new RussianPassport("1111", "567899");*

*System.out.println(document.getSeries()); // ошибка - тип Document ничего не знает о методах*

*RussianPassport passport = (RussianPassport) document; // явное приведение, теперь все ок*

*System.out.println(passport.getSeries());*

Для определения типа объекта используется метод ***instanceof***, возвращающий *boolean*. Например:

*Boolean isPasport = document instanceof RussianPassport;*

Термин **обобщение** (**дженерик**) означает параметризированный тип. Классы, интерфейсы и методы, оперирующие параметризированными типами, называются **обобщенными**. Это помогает алгоритмам работать одинаково, независимо от того, к данным какого типа они применяются (например, быстрая сортировка). Таким образом обобщения обеспечивают безопасность типов.

*class Gen<T> { // класс с обобщенным типом T*

*T ob;*

*Gen(T o) { // конструктор принимает объект типа T*

*ob = o; }*

*T getob() { // метод возвращает объект типа T*

*return ob; }*

*void showType() {*

*System.out.println(“Тип T – это “ + ob.getClass().getName()); } }*

*class GenDemo {*

*public static void main(String args[]) {*

*Gen<Integer> iOb; // создание ссылки на объект типа Gen<Integer>*

*iOb = new Gen<Integer>(88);*

*iOb.showType();*

*int v = iOb.getob();*

*Gen<String> strOb = new Gen<String>(“Тест”); // создание ссылки на объект типа Gen<String>*

*strOb.showType();*

*String str = strOb.getob(); } }*

В данном примере имя параметра ***T*** является **заполнителем**, подлежащим замене фактическим типом, передаваемым конструктору при создании объекта.

Механизм обобщений не позволяет использовать примитивные типы (int, double, char). Вместо них можно использовать любой тип класса (Integer, Double).

Обобщенный класс можно создать с несколькими параметрами типа:

*class TwoGen<T, V> { // класс с двумя обобщенными типами*

*T ob1;*

*V ob2;*

*Gen(T o1, V o2) {*

*ob1 = o1;*

*ob2 = o2; }*

*T getob1() {*

*return ob1; }*

*V getob2(){*

*return ob2; } }*

*class GenDemo {*

*public static void main(String args[]) {*

*TwoGen<Integer, String> tgObj = new TwoGen<Integer, String>(88, “Тест”);*

*Int v = tgObj.getob1();*

*String str = tgObj.getob2(); } }*

Когда необходимо ограничить параметризированные типы набором определенных типов используются **ограниченные типы**. Например, когда параметр *T* предполагает использование только числовых типов. Для информации, классы *Integer* и *Double* являются подклассами суперкласса *Number*.

*class NumericFns<T extends Number> { // аргументом типа должен быть Number или его подкласс*

*T num;*

*NumericFns(T n) { // конструктор принимает объект типа T*

*num = n; }*

*double reciprocal() { // метод возвращает объект типа double*

*return 1 / num.doubleValue(); } }*

*class BoundsDemo {*

*public static void main(String args[]) {*

*NumericFns <Integer> iOb = new NumericFns <Integer>(5);*

*System.out.println(“Обратная величина iOb - ” + iOb.reciprocal());*

*NumericFns <Double> dOb = new NumericFns <Double>(5.25);*

*System.out.println(“Обратная величина dOb - ” + dOb.reciprocal()); } }*

Для сравнения двух параметризированных типов, один из которых заранее неизвестен, используется **шаблон аргумента** (**?**). В данном случае выражение *NumericFns<?>* соответствует любому типу объекта из класса *NumericFns*:

*boolean absEqual(NumericFns<?> ob) {*

*if (Math.abs(num.doubleValue()) == Math.abs(ob.num.doubleValue()) {*

*return true; } }*

Шаблоны аргументов также можно ограничивать, как и параметры типов.

*static void test(Gen<? extends A> o) { // указать верхнюю границу (доступно A и всем его подклассам)*

*… }*

*static void test(Gen<? super B> o) { // указать нижнюю границу (доступно B и всем его суперклассам)*

*… }*

**Обобщенные методы** не требуют наличия обобщенного класса. Параметры типа объявляются перед возвращаемым типом. Например, метод сравнения двух массивов с одинаковыми или совместимыми типами:

*static <T extends Comparable<T>, V extends T> boolean arraysEqual(T[] x, V[] y) {*

*if (x.length != y.length) return false;*

*for (int i = 0; i < x.length; i++) {*

*if (!x[i].equals(y[i])) return false; }*

*return true; }*

**Обобщенные конструкторы** не требуют наличия обобщенного класса. Например, класс ниже вычисляет сумму всех чисел от 0 до *N*, при этом *N* передается обобщенному конструктору:

*class Summation {*

*private int sum;*

*<T extends Number> Summation(T arg) {*

*sum = 0;*

*for (int i =0; i <= arg.intValue(); i++) sum += i; }*

*int getSum() {*

*return sum; } }*

Любой класс реализующий **обобщенный интерфейс** должен быть также обобщенным. Исключение возможно только при указании интерфейсом конкретного типа. Обобщенные интерфейсы также могут быть ограниченными.

*interface Containment<T> {*

*boolean contains(T o); }*

*class MyClass<T> implements Containment<T> { // типизированный класс*

*… }*

*class MyClass implements Containment<T> { // нетипизированный класс*

*… }*

Обобщения позволяют использовать в качестве параметризированного типа тип ***Object***, но это небезопасно и может привести к ошибкам.

Вызов перегрузки методов обобщенных типов может вызывать неоднозначность, поэтому рекомендуется использовать методы с разными именами.

Начиная с JDK 7, можно сокращать объявление объектов обобщенного типа с помощью **ромбовидного оператора**:

*TwoGen<Integer, String> tgOb = new TwoGen<Integer, String>(42, “testing”); // полностью*

*TwoGen<Integer, String> tgOb = new TwoGen<>(42, “testing”); // сокращенно*

Ограничения использования обобщений:

1. Создать экземпляр параметра типа невозможно.

*class Gen<T> {*

*T ob;*

*Gen() {*

*ob = new T(); } } // нельзя*

1. В статическом члене нельзя использовать параметры типа, объявленные в его классе.

*class Wrong<T> {*

*static T ob; // нельзя*

*static T getob() { // нельзя*

*return ob; } }*

1. Нельзя получить экземпляр массива, тип элементов которого определяется параметром типа.

*vals = new T[10] // нельзя*

1. Нельзя создать массив обобщенных ссылок на объекты конкретного типа.

*Gen<Integer> gens[] = new Gen<Integer>[10] // нельзя*

1. Обобщенный класс не может расширять класс *Throwable*.

**Алгоритм** – это конечная последовательность действий для решения конкретной задачи. **Асимптотическая сложность** показывает, как количество операций, выполняемых алгоритмом, будет меняться в зависимости от объема входных данных. Всего существует около 20ти видов сложности. Сложность обозначается через O-нотацию. Виды сложности алгоритмов:

1. Линейная O(n). Например, найти единственный белый шар в мешке с черными шарами. Количество операций растет прямо пропорционально общему количеству шаров.
2. Логарифмическая O(log(n)). Например, поиск слова в словаре. Сложность тут растет медленнее.

**Бинарный (двоичный) поиск** основан на идее дихотомии – разделении набора объектов на два непересекающихся множества. Отсортированный массив разбивается на две части, и одна из них отбрасывается. Деление продолжается до тех пор, пока элемент не будет найден. Бинарный поиск есть в стандартной библиотеке *Arrays.binarySearch()*.

1. Константная (постоянная) O(1). Например, подойти к библиотекарю, спросить, где книга, и взять её.
2. Квадратичная O(n2). Например, найти дубликаты в массиве чисел.



**Рекурсия** – это функция, вызывающая сама себя. Она обязательно должна содержать **базовый случай** (условие выхода из рекурсии). Недостатки рекурсии:

* может замедлить работу программы по сравнению с циклами
* занимает много памяти
* может привести к переполнению стека и завершению программы с ошибкой

Пример рекурсии – подсчет факториала.

*int factR(int n) {*

*int result;*

*if (n == 1) return 1;*

*return factR(n-1) \* n; }*

Наиболее распространенные виды сортировок:

1. **Пузырьковая сортировка** (***buble sort***). Применяется только в учебных целях, т.к. существуют более эффективные алгоритмы. Массив обрабатывается в (n - 1) проходов. В каждой итерацию сравниваются соседние элементы и, при необходимости, меняются местами.

Сложность равна *O(n2)*. Является **стабильной сортировкой** (сохраняет относительный порядок элементов, что помогает при нескольких условиях сортировки).

*int[] array = { 10, 2, 10, 3, 1, 2, 5 };*

*boolean needIteration = true;*

*while (needIteration) {*

*needIteration = false;*

*for (int i = 1; I < array.length; i++) {*

*if (array[i] < array[i – 1]) {*

*int tmp = array[i];*

*array[i] = array[i – 1];*

*array[i – 1] = tmp;*

*needIteration = true; } } }*

1. **Сортировка слиянием** (***merge sort***). Рекурсивно разбивает задачи на меньшие, пока размер массива не достигнет единицы. Затем рекурсивно соединяет два соседних массива в упорядоченный третий массив.

Сложность равна *O(n \* log(n))*. Требуется, как минимум, *n* **дополнительной памяти**. Алгоритм **стабильной сортировки** (сохраняет относительный порядок равных элементов).

*public static void mergeSort(int[] array) {*

*if (array.length <= 1) {*

*return array; }*

*int[] result = new int[array.length]; // массив для результата сортировки*

*int[] left = mergeSortDescending(Arrays.copyOfRange(array, 0, array.length / 2));*

*int[] right = mergeSortDescending(Arrays.copyOfRange(array, array.length / 2, array.length));*

*int leftIdx = 0;*

*int rightIdx = 0;*

*int resultIdx= 0;*

*while (leftIdx < left.length && rightIdx < right.length) { // сливаем результаты*

*if (left[leftIdx] >= right[rightIdx]){*

*result[resultIdx] = left[leftIdx];*

*leftIdx++;*

*} else {*

*result[resultIdx] = right[rightIdx];*

*rightIdx++; }*

*resultIdx++; }*

*while (leftIdx < left.length) { // если один массив закончился раньше, чем второй*

*result[resultIdx] = left[leftIdx];*

*leftIdx++;*

*resultIdx++; }*

*while (rightIdx < right.length) {*

*result[resultIdx] = right[rightIdx];*

*rightIdx++;*

*resultIdx++; }*

*return result; }*

1. **Быстрая сортировка** (***quicksort***). Аналог пузырьковой сортировки. Выбирается опорный элемент (любой). Массив сортируется так, чтобы разбить его на две-три части (меньше опорного, равны опорному и больше опорного). Рекурсивно повторить к полученным частям.

Сложность равна *O(n \* log(n))*. Дополнительной памяти не требуется. Является **нестабильной сортировкой** (поддерживает только один порядок элементов).

*public static void quickSort(int[] source, int leftBoarder, int rightBoarder) {*

*int leftMarker = leftBoarder;*

*int rightMarker = rightBoarder;*

*int pivot = source[(leftMarker + rightMarker)] / 2];*

*do {*

*while (source[leftMarker] < pivot) {*

*leftMarker++; }*

*while (source[rightMarker] > pivot) {*

*rightMarker--; }*

*if (leftMarker <= rightMarker) {*

*if (leftMarker < rightMarker) {*

*int tmp = source[leftMarker];*

*source[leftMarker] = source[rightMarker];*

*source[rightMarker] = tmp; }*

*leftMarker++;*

*rightMarker--; }*

*} while (leftMarker <= rightMarker);*

*if (leftMarker <= rightBoarder) {*

*quicksort(source, leftMarker, rightBoarder); }*

*if (leftBoarder <= rightMarker) {*

*quicksort(source, leftBoarder, rightMarker); } }*

1. **Сортировка вставками** (***insertion sort***). Разделяет входной массив на две части (отсортированную и нет). Затем каждый элемент из неотсортированной части помещается на корректное место в отсортированной части со сдвигом массива.

Сложность равна *O(n2)*. Дополнительной памяти не требуется. Медленно работает на больших объемах.

*int[] array = { 10, 2, 10, 3, 1, 2, 5 };*

*for (int left = 0; left < array.length; left++) {*

*int value = array[left];*

*int i = left – 1;*

*for (; i >=0; i--) {*

*if (value < array[i]) {*

*array[i + 1] = array[i];*

*} else { break; } }*

*array[i + 1] = value; }*

1. **Поразрядная сортировка** (***radix sort***). Легкий вариант - считает общее количество вхождений каждого числового элемента и сохраняет его в массив-счетчик, а затем выводит его необходимое количество раз.

Тяжелый вариант - сравниваются значения одного крайнего разряда, и элементы группируются по результатам этого сравнения. Затем сравниваются значения следующего разряда, соседнего, и элементы либо упорядочиваются по результатам сравнения значений этого разряда внутри образованных на предыдущем проходе групп, либо переупорядочиваются в целом, но сохраняя относительный порядок, достигнутый при предыдущей сортировке. Затем аналогично делается для следующего разряда, и так до конца.

Сложность равна *O(n)*. Требуется дополнительная память, объёмом в размере количества уникальных значений.

Подробнее о сортировках [тут](https://javarush.com/groups/posts/1997-algoritmih-sortirovki-v-teorii-i-na-praktike).

**Автоупаковка** - процесс автоматической инкапсуляции простого типа данных в объектную оболочку соответствующего типа всякий раз, когда в этом возникает необходимость (создавать такой объект явным образом не нужно). **Автораспаковка** – это обратный процесс автоматического извлечения значения, упакованного в объектную оболочку. Автоматический процесс выглядит так:

*Integer iOb = 100;*

*int i = iOb;*

До версии JDK 5 упаковка и распаковка выполнялись вручную:

*Integer iOb = new Integer(100);*

*int i = iOb.intValue();*

Благодаря автораспаковке объектные оболочки целочисленных типов можно использовать в инструкции *switch*.

*switch (iOb) {*

*case 1: break;*

*case 2: break; }*

Следует помнить, что с любой операцией распаковки и упаковки связаны издержки, отсутствующие при использовании простых типов.

Преобразование строк для каждого из классов-обёрток:

*Byte.parseByte("12");*

*Short.parseShort("345");*

*Integer.parseInt("999999");*

*Long.parseLong("1000000000000");*

*Float.parseFloat("12.3");*

*Double.parseDouble("456.789");*

*Boolean.parseBoolean("true");*

**Структура данных** – это совокупность набора данных, отношений между ними и всех методов по управлению этими данными.



Интерфейс ***Iterable<T>*** предоставляет инструменты для того, чтобы перебирать (итерировать) коллекцию:

* Цикл *forEach*

*for (String element : list) {*

*System.out.println(element); }*

* Метод *iterator()*

*final Iterator<String> iterator; // объявили объект iterator*

*iterator = list.iterator(); // проинициализировали iterator через метод iterator()*

*while (iterator.hasNext()) { // проверяем, есть ли следующий элемент в коллекции*

*final String element = iterator.next(); // возвращаем элемент*

*System.out.print(element);}*

Интерфейс ***Collection<T>*** предоставляет инструменты для того, чтобы добавлять/удалять элементы, вычислять размер коллекции и осуществлять поиск внутри нее:

* Методы *add(E e)* и *addAll(Collection c)* для одного и нескольких элементов
* Методы *contains(Object o)* и *containsAll(Collection c)*
* Методы *remove(Object o)* и *removeAll(Collection c)*
* Метод *size()*
* Метод *isEmpty()*
* Другие полезные методы типа *clear()*, *toArray()* и т.д.

Интерфейс ***Set<T>*** позволяет реализовать структуры данных, где каждый элемент уникален (может встречаться только один раз).



**Множество** (***HashSet***) – это коллекция, в которой содержатся неупорядоченные уникальные элементы без индекса. Оно создает хеш-таблицу и использует только ячейки-ключи. Множество содержит методы:

* Метод ***add(E e)*** добавляет новый элемент в множество и возвращает значение *boolean*.
* Метод ***addAll(Collection<? Extends E> c)*** добавляет все элементы любой коллекции и возвращает значение *boolean*. Если хотя бы один из элементов будет успешно добавлен – вернется *true*.
* Метод ***contains(Object o)*** позволяет выяснить наличие объекта в множестве. Возвращает *boolean*.
* Метод ***remove(Object o)*** удаляет элемент из множества.
* Метод ***isEmpty()*** проверяет что множество пустое. Возвращает *boolean*.
* Метод ***size()*** возвращает количество элементов в множестве. Возвращает *int*.

Для реализации *HashSet* можно использовать следующие конструкторы:

* *public HashSet()* – стандартная реализация.
* *public HashSet(Collection<? Extends E> c)* – создает множество на основе списка или множества.
* *public HashSet(int initialCapacity, float loadFactor)* – задает начальное количество пустых ячеек и коэффициент заполнения.

Сложность поиска элемента в множестве всегда равна *O(1)*, т.к. оно всегда знает, где находится нужный элемент.

**Связанное множество** (***LinkedHashSet***) – это множество, построенное на основе хеш-таблицы *LinkedHashMap*. Для реализации *LinkedHashSet* можно использовать те же самые конструкторы, что и для *HashSet.* Сложность поиска элемента в связанном множестве всегда равна *O(1)*.

**Упорядоченное множество** (***TreeSet***) – это множество, построенное на основе хеш-таблицы *TreeMap*. Элементы должны реализовывать интерфейс *Comparable* (единственно верная сортировка для всех объектов класса) или при создании в конструктор нужно передать тип *Comparator* (экземпляр класса с логикой сортировки).

Для реализации *TreeSet* можно использовать следующие конструкторы:

* *public TreeSet()* – стандартная реализация.
* *public TreeSet(Comparator<? super E> comparator)* – реализация с логикой упорядочивания.
* *public TreeSet(Collection<? Extends E> c)* – создание на основе любого списка или множества.
* *public TreeSet(SortedSet<E> s)* – создание на основе уже отсортированного множества.

Сложность операций поиска, добавления и удаления элемента в упорядоченном множестве равна *O(log(n))*.

Интерфейс ***Queue<T>*** (**очередь**) описывает структуры данных, где элементы располагаются один за другим согласно очередности. Работает по принципу “первым зашел – первым вышел”.

Интерфейс ***List<T>*** позволяет создавать списки с повторяющимися элементами в произвольном порядке, которые могут расширяться и поддерживают сортировку. При этом он дает возможность контролировать процесс добавления нового элемента в список, организовать поиск и вносить изменения:

* Метод *add(E e)* добавляет элемент в конец списка
* Метод *add(int index, E e)* добавляет элемент в любое место списка
* Метод *get(int index)* получает элемент по индексу
* Метод *set(int index, E e)* заменяет элемент по индексу
* Метод *remove(int index)* удаляет элемент по индексу
* Метод *remove(E e)* удаляет элемент по значению
* Другие полезные методы

Список работает только со ссылочными типами, поэтому для примитивных типов нужно использовать класс-обертку.

Методы *Arrays.asList()* и *List.of()* позволяют создать список из массива или набора значений. Списки, созданные с помощью этих методов, считаются **неизменяемыми** (***ImmutableCollections***).

Метод *toArray(T[ ])* позволяет преобразовать список обратно в массив.

*List<String> cities = Arrays.asList("Воронеж", "Южно-Сахалинск", "Иркутск"); // создание списка*

*String[] citiesArray = cities.toArray(new String[ ]{ }); // преобразование списка в массив*

**Стек** – это набор элементов с организацией доступа по принципу “первым зашел – последним вышел”.

Список ***ArrayList<T>*** – это динамический массив, который может автоматически увеличиваться в размере во время исполнения программы. При вставке/удалении элемента, происходит сдвиг элементов вправо/влево.

У динамического массива есть два основных параметра – размер и вместимость. **Размер** – текущее количество элементов в списке. **Вместимость** – потенциальное количество элементов, которое может храниться в списке. По умолчанию равна 10 и автоматически увеличивается на 50% (в меньшую сторону) при превышении лимита.

*ArrayList<String> arrayListWithDefaultCapacity = new ArrayList<>(); // вместимость 10 элементов*

*ArrayList<String> arrayListWithSetCapacity = new ArrayList<>(33); // вместимость 33 элемента*

Сложность операции добавления элемента в конец списка ***add(T)*** в лучшем случае оценивается как *O(1)*, а в худшем – *O(n)*. В зависимости от необходимости увеличивать вместимость. **Амортизированная асимптотическая сложность** (когда список уже большой и увеличение происходит редко) равна *O(1)*.

Сложность операции добавления элемента по индексу ***add(int, T)*** в лучшем случае оценивается как *O(1)*, а в худшем – *O(n)*. В зависимости от количества сдвигаемых вправо элементов. Если список уже заполнен на момент вставки, то потребуется дополнительные *O(n)* на расширение. Аналогичная сложность с операциями удаления элемента ***remove(T)*** и по индексу ***remove(int)***.

Сложность операций получения элемента по индексу ***get(int)*** и замены элемента по индексу ***set(int, T)*** всегда оценивается как *O(1)*.

Список ***LinkedList*** реализует двунаправленный связанный список. Это набор связанных узлов, которые хранят ссылки на предыдущий и следующий элементы (как вагоны в поезде). Также двусвязный список хранит указатели на хвост и голову. Этот список поддерживает последовательный обход элементов друг за другом сначала или с конца.



Сложность операций добавления элемента в начало списка ***addFirst(T)*** и в конец списка ***addLast(T)*** всегда оценивается как *O(1)*. Их результатом становится перемещение ссылок головного и конечного элементов. Аналогичная сложность с операциями удаления из начала списка ***removeFirst()*** и из конца списка ***removeLast()***.

Сложность операций получения первого элемента ***getFirst()*** и последнего элемента ***getLast()*** всегда оценивается как *O(1)*.

Сложность операций добавления элемента по индексу ***add(int, T)***, получения элемента по индексу ***get(int)*** и удаления элемента по индексу ***remove(int)*** в лучшем случае оценивается как *O(1)*, а в худшем – *O(n)*. Чем ближе элемент расположен к центру списка, тем дольше его поиск.

Сложность операций получения размера списка ***size()*** всегда оценивается как *O(1)*, т.к. она динамически изменяется вместе со списком.

**Преимущества *ArrayList<T>*:**

1. Каждый новый элемент *ArrayList<T>* занимает меньше памяти, чем *LinkedList<T>*.
2. Массив *ArrayList<T>* расположен в памяти последовательно, в отличии от хаотично распределенных узлов *LinkedList<T>*. Повышается производительность.
3. Операции копирования массивов на практике работают быстрее, т.к. они часто оптимизированы на аппаратном уровне.

**Преимущества *LinkedList<T>*:**

1. В больших списках эффективнее работает со вставками/удалениями элементов в начале списка.
2. В больших списках эффективнее работает со вставками/удалениями элементов из середины списка.
3. Гарантирует вставку в конец списка за константное время *O(1)*.

Интерфейс ***Comparator<T>*** задает правила сортировки произвольных классов. Можно наследовать много раз. Для сравнения используется метод ***int compare(T, T)***, который возвращает:

* Отрицательное число, если первый объект меньше второго
* Ноль, если объекты равны
* Положительное число, если первый объект больше второго

Для сортировки коллекции в обратном порядке используется метод ***reverse()***.

Для каждого варианта сортировки нужно писать свой класс для сравнения. Например:

*import java.util.Comparator;*

*public class ItemPriceComparator implements Comparator<Item> { // на месте T - класс Item*

*@Override*

*public int compare(Item item1, Item item2) {*

*if (item1.price > item2.price) { // более дорогой должен быть дальше в списке*

*return 1;*

*} else if (item1.price < item2.price) { // более дешёвый — ближе к началу списка*

*return -1;*

*} else { // если стоимость равна, нужно вернуть 0*

*return 0; } } } // или упростить до строки return item1.price - item2.price;*

*public static void main(String[] args) {*

*List<Item> items = new ArrayList<>(); // создаём список товаров*

*items.add(new Item("Рубашка", 4500, 37));*

*items.add(new Item("Носки", 55, 8));*

*items.add(new Item("Толстовка", 1399, 74));*

*ItemPriceComparator itemPriceComparator = new ItemPriceComparator(); // создаём объект-компаратор*

*items.sort(itemPriceComparator); // сортировка по убыванию цены*

*System.out.println(items);*

*Comparator<Item> reversedItemPriceComparator = itemPriceComparator.reversed();*

*items.sort(reversedItemPriceComparator); // сортировка по возрастанию цены*

*System.out.println(items);}*

Для стандартных типов существуют готовые решения. Например, статическая переменная ***CASE\_INSENSITIVE\_ORDER*** хранит в себе *Comparator<String>*, сортирующий строки без учета регистра:

*static class ItemStringInSensitiveNameComparator implements Comparator<Item> {*

*@Override*

*public int compare(Item item1, Item item2) {*

*return String.CASE\_INSENSITIVE\_ORDER.compare(item1.name, item2.name); } }*

Классы-обертки не реализуют компараторы, но содержат статические методы ***compare()***. Например:

*public class ItemPriceComparator implements Comparator<Item> {*

*@Override*

*public int compare(Item item1, Item item2) {*

*return Integer.compare(item1.price, item2.price); } }*

В *Java* существует утилитарный класс ***Collections*** с более чем 20 методами для работы с различными структурами данных. **Утилитарные классы** не имеют конструкторов (либо они приватные), т.к. все его методы либо приватные, либо статические (доступны без создания экземпляра класса). **Утилитарные классы** не содержат изменяемых внутренних полей.

Следующие методы быстро создают неизменяемые списки нужного размера:

* Метод ***Collections.emptyList()*** создает пустой список.
* Метод ***Collections.singletonList(T)*** создает список из единственного элемента.
* Метод ***Collections.nCopies(int, T)*** создает список, где элемент *T* повторен *int* раз.

Превратить неизменяемый список в изменяемый можно с помощью передачи его в *ArrayList*:

*final List<String> immutableList = List.of("Ёж", "Медведь", "Лиса"); // неизменяемый список*

*final ArrayList<String> mutableList = new ArrayList<>(immutableList); // уже можно изменить*

Метод ***Collections.fill(List<T>, T)*** позволяет присвоить всем элементам списка *List<T>* единое значение *T*.

Метод ***Collections.copy(List<T>, List<T>)*** позволяет скопировать все элементы из второго списка в первый. Требует гораздо меньше ресурсов, чем пересоздание списка, если списки очень большие.

Метод ***Collections.sort(List<T>, Comparator<T>)*** позволяет сортировать списки (идентичен методу из *List<T>*).

Методы ***Collections.min(Collection, Comparator<T>)*** и ***Collections.max(Collection, Comparator<T>)*** помогают искать в списке минимум и максимум соответственно.

Интерфейс ***Comparable<T>*** задает естественную (единственно верную) сортировку для класса объектов. Его можно наследовать только один раз. Он предоставляет единственный метод ***int compareTo(T)***, который ведет себя аналогично методу *compare(T, T)*. С его помощью работают метод сортировки ***Collections.sort(List<T>)***, а также методы нахождения минимума ***Collections.min(Collection<T>)*** и максимума ***Collections.max(Collection<T>)***. Этот интерфейс лучше реализовывать для простых классов.

**Отображение** (***Map***) — это такое сопоставление, при котором объекту из первого множества соответствует только один объект из второго множества. Например, сопоставление множества стран и множества континентов можно записать кодом в виде хеш-таблицы *HashMap<>*.



Интерфейс ***Map<K, V>*** задает между элементами однозначную одностороннюю связь и позволяет работать с данными по принципу ключ-значение. Он предоставляет следующие методы:

* Метод ***put(K value, V value)*** добавляет новую запись. *K* – ключ, *V* – значение. Если ключ уже есть в хеш-таблице, то значение будет переписано.
* Методы ***containsKey(Object key)*** и ***containsValue(Object value)*** позволяют понять, содержится ли в хеш-таблице конкретный ключ или значение. Возвращают булево значение.
* Метод ***get(Object key)*** возвращает значение по ключу. Если ключ не найден, то вернет *null*.
* Метод ***getOrDefault(Object key, V defaultValue)*** возвращает значение по умолчанию, если не находит искомый ключ в хеш-таблице.
* Метод ***size()*** возвращает количество записей в хеш-таблице в типе *int*. Максимальное количество записей, которое можно хранить в хеш-таблице совпадает с макс. значением *int* и равно 2 147 483 647.
* Метод ***remove(Object key)*** возвращает значение, которое хранилось по этому ключу, и удаляет из хеш-таблицы соответствующую пару ключ-значение. Если значение или ключ отсутствуют, то вернется *null*.
* Метод ***clear()*** удаляет все данные из хеш-таблицы.
* Метод ***isEmpty()*** проверяет, хранится ли что-то в хеш-таблице (f*alse*) или она пустая (*true*). Возвращают булево значение.
* Метод ***keySet()*** итерирует по всем ключам.
* Метод ***values()*** итерирует по всем значениям.
* Метод ***entrySet()*** итерирует по всем записям. Возвращаемая запись ***Entry*** содержит одновременно ключ и значение. Доступ к ним осуществляется с помощью геттеров ***getKey()*** и ***getValue()***.

*for (Entry<String, Integer> entry : actorsMap.entrySet()) {*

*String actor = entry.getKey();*

*int filmsCount = entry.getValue(); }*

Хеш-таблицы работают только со ссылочными типами. В качестве значения можно передать и список.

Если в качестве ключа используется объект собственного класса, то для правильной работы хеш-таблицы нужно переопределить методы ***hasCode()*** и ***equals()***.

*@Override*

*public boolean equals(Object o) {*

*if (this == o) return true;*

*if (o == null || getClass() != o.getClass()) return false;*

*User user = (User) o;*

*return id.equals(user.id); }*

*@Override*

*public int hashCode() {*

*return id.hashCode(); }*

У каждого объекта в java есть метод ***hashCode()*** для его представления в виде числа. **Хеш-функция** (или ***функция-свёртки***) обеспечивает преобразование значения, возвращенного методом ***hashCode()***, в уникальное целое число - **хеш**. Хеш определяет, в какой ячейке хеш-таблицы будет сохранена запись.

Алгоритм хеш-функции обладает следующими свойствами:

* Детерминизм – для одного и того же ключа возвращается одинаковый хеш.
* Эффективность – должен работать быстро.
* Ограниченность – результат должен принадлежать определенному диапазону.
* Равномерность – данные должны быть распределены по ячейкам равномерно.

**Коллизия** – ситуация, при которой хеш-функция направляет два совершенно разных ключа в одну и ту же ячейку. В результате хеш-таблица продолжит корректно работать, но время её работы увеличится, т.к. придется перебирать все значения в ячейке на предмет соответствия вызываемому ключу. При некорректно реализованном методе *hashCode()*, все ключи скопятся в одной ячейке.

При увеличении количества ячеек, все ключи заново хешируются и перераспределяются. **Коэффициент заполнения** (***load factor***) – это соотношение между занятыми и свободными ячейками. По умолчанию он равен 75%, но его можно задать в момент создания хеш-таблицы:

*public HashMap(int initialCapacity, float loadFactor) // max initialCapacity = 1 073 741 824*

**Хеш-таблица** (***HashMap***) – это список пар ключ-значение, реализующий интерфейс ***Map***. Ключи уникальны, не генерируются автоматически и хранятся не упорядочено (при итерировании перебираются хаотично). Ячейки реализованы в виде массива длинной равной степени двойки (16, 32, 64 и т.д.). По умолчанию создаются 16 пустых ячеек и коэффициентом заполнения 75%.

Сложность базовых операций (поиск, добавление/обновление и удаление по ключу) всегда равна *O(1)*. Сложность итерации всегда будет равна *O(n)*.

**Связанная хеш-таблица** (***LinkedHashMap***) – это хеш-таблица, в которой ключи хранятся упорядоченно. Данные могут быть упорядочены двумя способами:

* В порядке добавления записей – при итерации первой будет запись, которая была добавлена в хеш-таблицу самой первой.
* В порядке доступа к записям – при итерации первой будет запись, к которой дольше всего не обращались. Обращением считается вызов одного из методов выборки по ключу – *get(Object key)* и *getOrDefault(Object key, V defaultValue)*.

По умолчанию создаются 16 пустых ячеек и коэффициентом заполнения 75%. Для создания новой связанной хеш-таблицы можно использовать следующие конструкторы:

* *public LinkedHashMap()* – стандартные настройки, сортировка в порядке добавления.
* *public LinkedHashMap(Map<? Extends K, ? extends V> m)* – копирование хеш-таблицы *m*.
* *public LinkedHashMap(int initialCapacity)* – стандартные настройки с созданием *int* ячеек.
* *public LinkedHashMap(int initialCapacity, float loadFactor) –* стандартные настройки с созданием *int* ячеек и указанием коэффициента заполнения *float*.
* *public LinkedHashMap(int initialCapacity, float loadFactor, Boolean accessOrder)* – стандартные настройки с созданием *int* ячеек, указанием коэффициента заполнения *float* и типом сортировки (*true* – по порядку добавления записей, *false* – в порядке доступа к записям).

Сложность базовых операций (поиск, добавление/обновление и удаление по ключу) всегда равна *O(1)*. Из-за наличия дополнительного двусвязного списка и накладных расходов на его поддержание, связанная хеш-таблица потребляет чуть больше памяти и совершает чуть больше действий, чем простая хеш-таблица. Сложность итерации всегда будет равна *O(n)*.

**Упорядоченная хеш-таблица** (***TreeMap***) – это хеш-таблица, в которой применяются сложные правила сортировки элементов (по заданному признаку). Например, для строк это “от первой буквы алфавита к последней”, а для чисел – “от меньшего к большему”. Наличие четкого порядка упрощает итерацию и создает полезные возможности.

Для создания новой упорядоченной хеш-таблицы можно использовать следующие конструкторы:

* *public TreeMap()* – создает пустую хеш-таблицу без пустых ячеек.
* *public TreeMap(Comparator<? super K> comparator)* – создает хеш-таблицу и задает правила её сортировки.
* *public TreeMap(Comparator<? extends K, ? extends V> m)* – создает новый объект *TreeMap* со стандартными настройками и заполняет данными из переданной хеш-таблицы любого типа.
* *public TreeMap(SortedMap<K, ? extends V> m)* – создает новый объект *TreeMap* со стандартными настройками и заполняет данными из переданной хеш-таблицы типа *SortedMap* (с уже отсортированными данными).

Например, зададим порядок сортировки при передаче в конструктор объекта *Comparator*. Обычно этот метод применяется, когда нельзя отредактировать код в исходном классе, или когда в разных местах нужна разная логика сортировки.

*Comparator<User> userComparator = new Comparator<>() {*

*@Override*

*public int compare(User user1, User user2) {*

*return user1.userId - user2.userId; } };*

*Map<User, String> users = new TreeMap<>(userComparator);*

Например, зададим порядок сортировки через интерфейс *Comparable*. Этот метод удобен, когда есть только один способ сортировки данных.

*class User implements Comparable<User> {*

*public Integer userId;*

*public String username;*

*@Override*

*public int compareTo(User o) {*

*return this.userId - o.userId; } }*

Методы ***firstEntry()*** и ***lastEntry()*** возвращают первую и последнюю по порядку запись.

Методы ***firstKey()*** и ***lastKey()*** возвращают первый и последний по порядку ключ.

Методы ***pollFirstEntry()*** и ***pollLastEntry()*** возвращают первую и последнюю по порядку запись, а также удаляют их из хеш-таблицы.

Сложность базовых операций (поиск, добавление/обновление и удаление по ключу) всегда равна *O(log(n))*. Это ненамного медленнее чем *O(1)*, но выигрышнее за счет сортировки. Для поддержания сортировки упорядоченная хеш-таблица потребляет чуть больше памяти и совершает чуть больше действий, чем простая хеш-таблица. Сложность итерации всегда будет равна *O(n)*.

На примере иерархии сотрудников рассмотрим из чего состоят деревья.



* **Вершина дерева** (***vertex***), или **узел/нода** — объект, который может хранить информацию и ссылаться на другие вершины. Может иметь не более одного родителя (руководителя). У вершины могут быть потомки (подчиненные). Если потомков нет, то она называется **листом** (***leaf***). В иерархии организации вершинами являются все сотрудники.
* **Ребро дерева** (***edge***) — направленная, то есть иерархическая, связь между двумя вершинами. Направление рёбер дерева однозначно определяет какая из двух связанных вершин выше по иерархии, а какая — ниже. В иерархии организации Рита подчиняется Ёлкину.
* **Путь в дереве** (***path***), или **маршрут** — набор вершин, в котором следующие друг за другом узлы соединены рёбрами. Так как рёбра имеют строго одно направление, то и путь можно проложить только в одну сторону — сверху вниз по иерархии и никак иначе. Количество вершин в самом длинном пути от корня до листа называется **высотой**/**глубиной** древа с корнем.
* **Дерево** — совокупность вершин и рёбер без циклов. Если из вершины X1​ существует путь до вершины X2​, то гарантируется, что не существует пути от X2​ к X1​. Это значит, что сотрудник не может быть одновременно и подчинённым, и руководителем для одного и того же человека.

Если мы ограничим количество дочерних узлов в вершинах дерева числом *N*, то это число будет называться **арностью дерева**. Если арность равна двум, то дерево называют **бинарным** или **двоичным**.

Чтобы бинарное дерево превратилось в **дерево поиска**, оно должно отвечать нескольким условиям:

1. Если в узле дерева записано значение Х, то в левом поддереве располагаются только узлы со значениями, меньшими или равными Х.
2. Если в узле дерева записано значение Х, то в правом поддереве располагаются только узлы со значениями, большими или равными Х.
3. Левое и правое поддеревья отвечают тем же двум условиям. Это рекурсивное правило. Из него следует, что любое поддерево в дереве поиска будет являться деревом поиска.

Если в процессе поиска обнаруживается, что отсутствует ссылка на нужное поддерево, то искомого элемента не существует.

**Исключение** – это ошибка, возникающая в процессе выполнения программы.

**Обработчик исключений** – это автоматизированный код, выполняющийся при возникновении ошибки.



Все исключения являются потомками класса ***Throwable***. У него есть два подкласса – ***Exception*** (ошибки программы) и ***Error*** (ошибки машины java и нехватки памяти). Проверяемые исключения обязательны к обработке (иначе код не скомпилируется) и наследуются от ***Exception***, а непроверяемые – на усмотрение разработчика и наследуются от ***RuntimeException*** или ***Error***.

**Стек-трейс** (***stack trace***) – сообщение об ошибке с информацией о месте и причине её появления. Он реализует логику последним вошел, первым вышел.

Основным блоком обработки исключений является инструкция ***try-catch***. Когда в блоке *try* возникает исключение, оно перехватывается соответствующей инструкцией *catch* и обрабатывается в соответствии с ней. Одному блоку *try* могут соответствовать множество блоков *catch* (для разных типов исключений).

Для выполнения операций, связанных с нормальным завершением программы, используется инструкция ***finally***. Она выполняется всегда и не зависит от наличия необработанных исключений. Не рекомендуется использовать в блоке *finally* инструкцию *return*.

*try {*

*// код, из-за которого может произойти исключение }*

*catch (Throwable throwable) { // параметр - тип возможного исключения*

*// код, который выполнится, если произойдёт исключение указанного типа }*

*finally {*

*// необязательный блок, который выполнится в любом случае }*

Условие перехвата исключений суперкласса справедливо и для его подклассов. Исключения обрабатываются в порядке их расположения в программе, поэтому если указать первым тип *Throwable* (суперкласс всех исключений), то все остальные блоки обработки исключений проигнорируются.

Метод ***getMessage()*** возвращает краткое описание ошибки, которое можно вывести/записать.

Метод ***printStackTrace()*** выводит на экран стек-трейс об ошибке.

Метод ***getStackTrace()*** возвращает массив элементов повлекших вызов исключения. С помощью дополнительных методов ***getClassName()***, ***getMethodName()***, ***getFileName()*** и ***getLineNumber()*** можно получить конкретную информацию для каждого элемента массива.

Блоки *try* могут быть вложенными один в другой. Если исключение во внутреннем блоке не будет отработано, то оно распространится на внешний блок.

Для ручной генерации исключений используется инструкция ***throw***. Объект исключения должен быть объектом класса, производного от *Throwable*. Сформированный при этом объект исключения доступен для повторной генерации исключений.

*throw new ArithmeticException();*

Для создания собственного исключения необходимо создать подкласс, производный от ***Exception***, который порожден классом *Throwable*. В новый подкласс необязательно добавлять реализацию каких-либо методов – его можно использовать в качестве исключения сразу по создании.

*public class NonlntResultException extends Exception {*

*int n;*

*int d;*

*public NonlntResultException(int i, int j) { // пишем свой конструктор исключения*

*n = i;*

*d = j; }*

*public String toString() {*

*return "Результат операции " + n + " / " + d + “ не является целым числом"; } }*

*public class CustomExceptDemo {*

*public static void main(String args[]) {*

*int numer[] = { 4, 8, 15, 32, 64, 127, 256, 512 };*

*for(int i=0; i < numer.length; i++) {*

*try {*

*if((numer[I ]% 2) != 0)*

*throw new NonlntResultException(numer[i], denom[i]); }*

*catch (NonlntResultException exc) {*

*System.out.println(exc); } } } }*

Инструкция ***throws*** позволяет обрабатывать исключения вне метода, в котором они возникают (его отлавливает вышестоящий метод или JVM). Он может использоваться в методах и конструкторах.

*public void methodWithException() throws FirstException { // могут указываться через запятую*

*if (какое-то условие) {*

*throw new FirstException(); } } // сгенерировать исключение*

Начиная с JDK 7 блок *try* дополнительно поддерживает:

1. Автоматизированный процесс освобождения ресурсов (например файлы).
2. **Групповой перехват** нескольких исключений одной инструкцией *catch*. Для этого неявно используется ключевое слово final и явно побитовый оператор ИЛИ.

*catch (final ArithmeticException | ArrayIndexOutOfBoundsException e) {*

*// блок кода }*

1. **Окончательное (или утонченное) повторное генерирование** **исключений**, которое ограничивает процесс лишь теми проверяемыми типами исключений, которые генерируются в блоке *try* и не обрабатываются в предыдущих блоках *catch*, а также относится к подтипу или супертипу указываемого параметра. Для этого используется ключевое слово ***final***.

Для представления файлов в Java существуют два класса – ***java.io.File*** (старый) и ***java.nio.file.Path*** (начиная с 7 версии).

**Абсолютный путь** – это полный путь, начиная с корневого каталога.

**Относительный путь** – это путь относительно какой-либо отправной точки (например, текущей).

Старый класс ***File*** использовался для управления информацией о файлах и директориях. У него есть два конструктора *File(File parent, String child)* и *File(String parent, String child)*.

Метод ***getName()*** возвращает строку с именем файла или директории.

Метод ***isFile()*** возвращает булево значение определяющее наличие файла.

Метод ***isDirectory()*** возвращает булево значение определяющее наличие директории.

Метод ***list()*** возвращает массив имен файлов и поддиректорий *String[]*.

Метод ***listFiles()*** возвращает массив объектов файлов и поддиректорий *File[]*.

Метод ***mkdir()*** создает новую директорию и возвращает *true* при успешном создании.

Метод ***renameTo(File)*** переименовывает файл или директорию на имя в параметре. Возвращает *false* при неудаче.

Метод ***delete()*** удаляет файл или пустую директорию и возвращает *true* при успехе.

В разных системах используется разные разделительные знаки, поэтому в классе *File* есть константа ***separator***, которая автоматически адаптируется под операционную систему.

*File dir = new File(HOME + File.separator + "NewDir");*

У класса *File* отсутствуют методы копирования и перемещения файлов, и слишком много методов возвращают булево значение, поэтому его переработали в 7й версии *Java*. В новой версии появилось три структуры:

1. Класс ***Paths*** – служебный класс с двумя статическими методами get для получения объекта типа Path. Методы отличаются только параметрами.

Метод ***get(String first, String… more)*** преобразовывает строку пути или последовательность строк образующих при соединении строку пути в *Path*.

Метод ***get(URI uri)*** преобразовывает символьную строку на ресурс (*file:///C:/books/dune.epub*) в объект *Path*.

1. Интерфейс ***Path*** – доработанная версия *File*, у которого есть множество реализаций для конкретных файловых систем. Он содержит методы для добавления элементов пути, их извлечения и манипуляций с ними.

Метод ***Path getFileName()*** возвращает имя файла из пути.

Метод ***Path getParent()*** возвращает родительскую директорию по отношению к текущему пути.

Метод ***Path getRoot()*** возвращает корневую директорию.

1. Класс ***Files***, состоящий из множества статических методов для работы с файлами.

Метод ***Path createFile(Path path, FileAttribute<?>… attrs)*** создает новый пустой файл. Если файл существует – выбрасывает исключение. Необязательный список атрибутов *attrs* может содержать правила доступа к файлу, информацию о создателе и т.д.

Метод ***Path createDirectory(Path dir, FileAttribute<?>… attrs)*** создает новую директорию.

Метод ***Path move(Path source, Path target, CopyOption… options)*** перемещает файл.

Метод ***void delete(Path path)*** удаляет файл или пустую директорию. Если директория не пуста или файла не существует – выбросит исключение.

Метод ***boolean deleteIfExists(Path path)*** удаляет файл или директорию, если они существуют.

Метод ***Path copy(Path source, Path target, CopyOption… options)*** копирует указанный объект. При копировании директории содержащиеся в ней файлы и каталоги не будут скопированы. В *options* могут быть до трех параметров:

* *REPLACE\_EXISTING* – заменить файл, если он уже существует в директории назначения.
* *COPY\_ATTRIBUTES* – скопировать атрибуты оригинального файла в его копию.
* *ATOMIC\_MOVE* – переместить файл **атомарно** (либо все, либо ничего).

**Поток** – это бесконечная последовательность данных. Поток подключен к источнику *source*и получателю *destination*. В *java* определены **байтовые** (байты, они же символы в коде *ASCII*) и **символьные** потоки (символы в коде *Unicode*). Символьные потоки являются адаптированными байтовыми потоками.

Все необходимые классы для работы с потоками ввода-вывода находятся в пакете *java.io*. Для каждого из типов поток *Java* предлагает отдельный базовый абстрактный класс.



Помимо этих потоков, в программы автоматически импортируется пакет *java.lang* с классом *System*, в котором определены переменные ***in***, ***out*** и ***err***, представляющие стандартные байтовые потоки ввода-вывода. Они являются экземплярами *InputStream* (для первого) и *PrintStream* (для остальных) и поддерживают их методы.

Для работы с файлами у каждого из четырех абстрактных классов потоков есть своя реализация:

* *FileInputStream* и *FileOutputStream* лучше использовать для бинарных файлов (картинки, видео, pdf).
* *FileReader* и *FileWriter* лучше использовать для текстовых файлов, хотя можно применять и байтовый поток.

Общая схема работы с потоками и файлами:

1. Создается потоковый объект и ассоциируется с фалом на диске
2. Данные читаются из потока или записываются в поток
3. Поток закрывается.

При возникновении ошибок чтения/записи потоком выдается исключение типа ***IOException***.

У класса *FileWritter* существует несколько конструкторов. Первый ***FileWriter(String)*** – переписывает файл каждый раз заново, второй ***FileWriter(String, boolean)*** – при передаче *true* добавляет данные в конец файла.

*public static void main(String[] args) throws IOException {*

*Writer fileWriter = new FileWriter("filewriter.txt"); // откроем поток*

*fileWriter.write("новая запись в новый файл\n"); // добавим строку*

*fileWriter.close(); } // закроем поток*

Класс *FileReader* считывает данные по одному символу пока не достигнет конца файла. Метод int read() возвращает значение char прочитанного символа. Если он вернет -1, значит файл закончился.

*public static void main(String[] args) throws IOException {*

*Reader fileReader = new FileReader("filereader.txt");*

*int data = fileReader.read();*

*while (data != -1) {*

*System.out.print((char) data);*

*data = fileReader.read(); }*

*fileReader.close(); }*

В примерах выше класс *FileReader* обращался за каждым символом напрямую к файлу. Этот процесс можно ускорить, применив **буферизацию** – способ ввода/вывода данных, при котором для их временного хранения используется область памяти (**буфер**). При создании буфера можно указать его размер. Например, в классе ***BufferedReader*** по умолчанию используется 8192 символа.

*public static void main(String[] args) throws IOException {*

*FileReader reader = new FileReader("bufferedreader.txt");*

*BufferedReader br = new BufferedReader(reader); // буферизация*

*while (br.ready()) {*

*String line = br.readLine();*

*System.out.println(line); }*

*br.close(); }*

Начиная с JDK 7, появился автоматизированный процесс освобождения ресурсов конструкцией ***try-with-resources***. Под ресурсами понимается любой класс, наследуемый от интерфейсов ***Closeable*** или ***AutoCloseable***. При необходимости можно закрыть доступ явным образом через метод *close()*. В блоке *try-with-resources* можно объявить несколько ресурсов, разделенных точкой с запятой, однако ресурсы, открытые первыми, будут закрыты последними.

*public class Practicum {*

*public static void main(String[] args) {*

*try (Resource1 resource1 = new Resource1(); Resource2 resource2 = new Resource2()) {*

*System.out.println("внутри блока try"); } } }*

*class Resource1 implements AutoCloseable { // создаем свой класс*

*@Override*

*public void close() { // и реализуем интерфейс*

*System.out.println("метод close() для Resource1"); } }*

**Кодировка** – набор числовых значений, в котором каждое значение соответствуют своему символу.

Кодировка ***ASCII*** (***American Standart Code for Information Interchange***) определяет 128 символа от 0 до 127. Таблица начинается с первых 32 невидимых управляющих символов и заканчивается невидимым управляющим символом *DEL*. Между ними располагаются видимые пробел, знаки препинания, латинские буквы и цифры.

Кодировка ***Unicode*** разбит на 17 плоскостей, содержащих по 65536 числовых значений (всего 1 114 112). Она включает все современные алфавиты, цифры, знаки препинания, математические, денежные, музыкальные символы и символы мертвых языков. Самый распространенный вариант – 8-битный ***UTF-8***.

Для представления кодировок в *java* существует специальный класс ***Charset*** и пакет *java.nio.charset.StandardCharsets*. Задать кодировку можно либо с помощью класса ***StandardCharset*** с константами для стандартного набора символов, либо через статический метод ***Charset*** ***forName(String charsetName)***. Первый способ предпочтительнее.

Если не задать кодировку явно, то будет использована кодировка по умолчанию (на основании выбранного языка и операционной системы). Определить кодировку по умолчанию можно так:

*System.out.println(Charset.defaultCharset().displayName());*

Проблему кодировки по умолчанию решает запуск *Java* с параметром. Например:

*java -Dfile.encoding=UTF-8 Practicum*

Для основных классов поток предусмотрены конструкторы, в которые можно передать кодировку. Например, ***InputStreamReader(InputStream in, Charset cs)***, ***PrintStream(boolean autoFlush, OutputStream out, Charset charset)*** и другие.

Многие методы могут работать только с массивом байтов, поэтому есть соответствующие методы ***getBytes()***, преобразующий строку в байтовый массив, и ***String(byte bytes[])***, создающий строку из байтового массива. В них тоже можно указывать кодировку в качестве параметра.

**Функциональный стиль программирования** заключается в идее обращения с действиями как с объектами, которые можно сохранять в переменные и передавать в методы в качестве аргументов. Это помогает избежать изменения уже написанного кода класса и помогает избежать новых багов.

Например, пусть конструктор класса принимает объект интерфейса (как внешний аргумент функции), который реализует функционал преобразования строк. В этом случае, для переключения между реализациями не потребуется изменять уже написанный класс, а всего лишь передать нужный объект интерфейса.

*interface StringsTransformConfig {*

*String transform(String line); }*

*class StringsTransform {*

*private List<String> saved = new LinkedList<>();*

*private StringsSaverConfig config;*

*public StringsSaver(StringsTransformConfig config) {*

*this.config = config; }*

*public void save(String line) {*

*line = config.transform(line); }*

*public List<String> getSaved() return saved; }*

*class MyConfig implements StringsTransformConfig {*

*@Override*

*public String transform(String line) return "[" + LocalDateTime.now() + "] " + line; }*

**Анонимный класс** – специальный инструмент, позволяющий объявить одноразовый безымянный класс в момент создания его объекта. Для этого после объявления переменной и знака равно указывается родительский класс (или имплементируемый интерфейс), круглые скобки и фигурные скобки с реализацией. Внутри анонимных классов могут быть нестатические поля и методы. Все локальные переменные и параметры вне анонимного класса доступны ему только для чтения (они не должны потом изменяться в коде).



**Функциональный интерфейс** — это интерфейс, который содержит только один абстрактный метод (например, *Runnable*). Для его обозначения используется аннотация ***@FunctionalInterface***. Начиная с JDK 8, допускается чтобы функциональный интерфейс имел **методы по умолчанию**, которые не являются абстрактными, и статические методы.

Таким образом для реализации функционального стиля, объекты интерфейса можно передавать через сеттеры с анонимными классами, реализующими функциональные интерфейсы.

Функциональные интерфейсы могут быть обобщенными:

*interface SomeTest<T> {*

*boolean test(T n, T m); }*

*class GenericFunctionalInterfaceDemo {*

*public static void main(String args[]) {*

*SomeTest<Integer> isFactor = (n, d) -> (n % d) == 0;*

*if (isFactor.test(10, 2)) System.out.println(“2 является делителем 10”);*

*SomeTest<Double> isFactorD = (n, d) -> (n % d) == 0;*

*if (isFactorD.test(212.0, 4.0)) System.out.println(“4.0 является делителем 212.0”);*

*SomeTest<String> isIn = (a, b) -> a.indexOf(b) != -1;*

*String str = “Обобщенный функциональный интерфейс”;*

*if (isIn.test(str, “face”)) System.out.println(“’face’ найдено”); }*

Функциональные интерфейсы из стандартной библиотеки *java.util.function*:

* ***Consumer<T>*** содержит метод *accept(T)*, который ничего не возвращает. Удобен для вывода в консоль.
* ***Supplier<T>*** содержит метод *get()*, который возвращает объект типа *T*. Удобен для генератора чисел.
* ***Predicate<T>*** содержит метод *test(T)*, который возвращает булево значение. Удобен для проверок на *null*.
* ***Function<T, R>*** содержит метод *apply(T)*, который возвращает объект типа *R*. Удобен для парсинга числа из строки.
* ***UnaryOperator<T>*** содержит метод *apply(T)*, который возвращает объект типа *T*. Удобен для модернизации объектов.

**Лямбда-выражение** – специальный инструмент, позволяющий быстро и компактно имплементировать функциональный интерфейс. Оно появилось с *JDK* 8 и состоит из двух частей:

*(список входных параметров) -> { блок реализации функционального интерфейса }*

*() -> 98.6; // лямбда-выражения возвращает число*

*double myMeth() { return 98.6; } // обычный метод, возвращающий число*

Правила синтаксиса лямбда-выражений:

* У лямбды может быть 0 и более входных параметров.
* Параметры заключаются в круглые скобки и разделяются запятыми.

Например, *(String country, int size)*.

* Если Java может получить тип параметра из контекста, его можно не указывать.

Например, *(country, size)*.

* Если параметр один и его тип явно не указывается, скобки можно не писать.

Например, *distance -> return distance / 100*.

* Если параметров нет, нужно поставить пустые круглые скобки.

Например, *() -> "магазин закрыт"*.

* Тело лямбды может состоять из 0 и более выражений.
* Если тело состоит из одного оператора, его можно не заключать в фигурные скобки. Возвращаемое значение в этом случае можно указать без ключевого слова *return*.

Например, *distance -> distance / 100*.

* Во всех остальных случаях фигурные скобки обязательны, а в конце блока кода необходимо указать возвращаемое значение с использованием ключевого слова *return* (иначе будет возвращать *void*).

Сравним применение функционального интерфейса через анонимный класс и лямбда-выражение. Как видим, последнее выглядит более компактно и лаконично. Однако лямбды имеют более узкое применение, чем анонимные классы, т.к. не могут реализовывать несколько абстрактных методов и не поддерживают поля.

*public static void main(String[] args) {*

*Consumer<String> greeter = new Consumer<>() {*

*@Override // реализация через анонимный класс*

*public void accept(String name) {*

*System.out.println("Привет, " + name + "!"); } };*

*greeter.accept("Дарья"); }*

*public static void main(String[] args) { // реализация через лямбда-выражение*

*Consumer<String> greeter = name -> System.out.println("Привет, " + name + "!");*

*greeter.accept("Дарья"); }*

Лямбда-выражение имеет доступ к полям объекта, внутри которого она создана. Благодаря этому она реализует механизм **замыкания** – при изменении значения поля метод лямбды будет работать с новым значением этого поля. Лямбда также может менять эти поля.

Локальные переменные и параметры внутри лямбда-выражения называются **эффективно финальными**, т.е. неизменными.

Для модификации списка можно использовать метод **removeIf()**, который удаляет элементы, подошедшие под указанное условие. Рассмотри пример удаления пустых строк в списке. В таких случаях можно использовать **ссылки на методы**, позволяющие ссылаться на метод (статический и нестатический) без его выполнения.

*List<String> countries = new ArrayList<>(List.of("Мозамбик", "Япония", "", "США", ""));*

*countries.removeIf(name -> name.isEmpty()); // лямбда*

*countries.removeIf(String::isEmpty); // ИмяКласса::метод*

**Ссылка на метод экземпляра**. Проверим содержится ли указанный объект в списке.

*Predicate<String> hasCountry = str -> countries.contains(str); // лямбда*

*Predicate<String> hasCountry = countries::contains; // объект::метод*

**Ссылка на конструктор**. Метод *apply* функционального интерфейса *Function* принимает число и возвращает *ArrayList* (отсылка к конструктору с вместимостью списка).

*Function<Integer, ArrayList<Integer>> listCreator = capacity -> new ArrayList<>(capacity); // лямбда*

*Function<Integer, ArrayList<Integer>> listCreator = ArrayList::new; // Класс::new*

Лямбда-выражения могут генерировать исключения совместимые с исключениями реализуемого абстрактного метода.

Для проверки защиты от исключения *NullPointerException* можно использовать класс-обертку ***Optional<T>***. У него есть следующие методы:

* Метод ***isPresent()*** возвращает *true*, если внутри не *null*.
* Метод ***ifPresent()*** позволяет выполнить какое-либо действие, если объект не пустой.
* Метод ***get()*** возвращает распакованное значение.
* Метод ***orElse(value)*** в случае отсутствия внутри значения вернет указанный параметр.
* Метод ***orElseThrow()*** в случае отсутствия значения выбросит исключение.
* Метод ***Optional.of(value)*** создает класс-обертку с указанным параметром.
* Метод ***Optional.empty()*** создает пустой класс-обертку.

Для примитивных типов также существуют классы-обертки. Например, ***OptionalInt*** и ***OptionalDouble***. Для некоторых примитивов есть собственные типизированные стримы – ***IntStream.range(int, int)***.

Подробнее можно прочитать [тут](https://habr.com/ru/post/346782/).

Дженерик-интерфейс ***Stream*** – встроенный инструмент *java* для потоковой обработки данных в функциональном стиле. Стрим можно создать через статические методы ***Stream.of()***, ***Stream.iterate()*** либо через метод коллекции ***stream()***. Также можно получить стрим из строк файла через ***Files.lines(Path path)***.

*Stream<String> names = Stream.of("Арсений", "Иван", "Аня");*

*List<String> citiesToVisit = List.of("Лондон", "Мадрид", "Владивосток");*

*Stream<String> visitStream = citiesToVisit.stream();*

Операции над стримами бывают двух видов – промежуточные и терминальные.

**Терминальные** – конечная операция, вызывающаяся после промежуточных. Вызывается один раз.

* Операция ***forEach()*** применяет переданный в нее параметр ко всем элементам стрима. Например, можно вывести в консоль все, что передано в стрим.

*Stream.of("фильм «Компиляторы-убийцы»", "сериал «Спринг»", "сериал «Алгоритмы»")*

*.forEach(System.out::println);*

* Операция ***count()*** считает число элементов стрима (после всех фильтров).

*long count = Stream.of("фильм «Компиляторы-убийцы»", "сериал «Спринг»", "сериал «Алгоритмы»")*

*.count();*

* Операция ***allMatch()*** проверяет, что все ли элементы стрима удовлетворяют условию-параметру.

*boolean isAllFilm = Stream.of("фильм «Компиляторы-убийцы»", "сериал «Спринг»", "сериал «Алгоритмы»")*

*.allMatch(title -> title.startsWith("фильм"));*

* Операция ***anyMatch()*** проверяет, что хотя бы один элемент удовлетворяют условию-параметру.

*boolean found = Stream.of("фильм «Компиляторы-убийцы»", "сериал «Спринг»", "сериал «Алгоритмы»")*

*.anyMatch(title -> title.toLowerCase().contains("спринг"));*

* Операция ***noneMatch()*** проверяет, что все элементы не удовлетворяют условию-параметру.

*boolean found = Stream.of("фильм «Компиляторы-убийцы»", "сериал «Спринг»", "сериал «Алгоритмы»")*

*.noneMatch(title -> title.toLowerCase().contains("тест"));*

* Операция ***findFirst()*** возвращает первый элемент стрима.

*Optional<String> maybeFound = Stream.of("сериал «Спринг»", "сериал «Алгоритмы»", "фильм «Матрица»")*

*.findFirst();*

* Операция ***findAny()*** возвращает любой элемент стрима.

*Optional<String> maybeFound = Stream.of("сериал «Спринг»", "сериал «Алгоритмы»", "фильм «Матрица»")*

*.findFirst();*

* Операция ***collect()*** позволяет собрать элементы стрима в коллекцию или другой объект.

*List<String> collected = Stream.of("сериал «Спринг»", "сериал «Алгоритмы»", "фильм «Матрица")*

*.collect(Collectors.toList());*

* Операции ***min()*** и ***max()*** позволяют найти минимальный и максимальный элементы стрима.
* Операция ***sum()*** позволяют найти сумму элементов стрима.
* Операция ***average()*** позволяют найти среднее значение элементов стрима.
* Операция ***summaryStatistics()*** считает всю статистику стрима (*count, sum, min, max, average*).
* Операция ***reduce()*** позволяет собрать элементы стрима в один поток (обратный аналог *flatMap()*).

*int reduceResult = Stream.iterate(1, i -> i <= 5, i -> i + 1)*

*.reduce(1, (acum, cur) -> acum \* cur); // рассчет факториала от 1 до 5*

**Промежуточные** – задают правила изменения стрима, возвращая трансформированный поток исходных данных. Например, фильтрация. Может быть задано несколько промежуточных операций.

* Операция ***filter()*** возвращает новый стрим из элементов удовлетворяющих заданному предикату.

*Stream.of("фильм «Компиляторы-убийцы»", "сериал «Спринг»", "сериал «Алгоритмы»")*

*.filter(title -> title.contains(“фильм”)*

*.forEach(System.out::println);*

* Операция ***map()*** применяет лямбду к элементам стрима и возвращает новый стрим. Каждому элементу исходного стрима соответствует один элемент нового стрима.

*Stream.of("фильм «Компиляторы-убийцы»", "сериал «Спринг»", "сериал «Алгоритмы»")*

*.map(ratings::get) // значение берется из хэшмапы ratings*

*.forEach(System.out::println);*

* Операция ***flatMap()*** применяет лямбду к элементам стрима и возвращает новый стрим. Каждому элементу исходного стрима соответствует 0, 1 или несколько элементов нового стрима.

*Stream.of("ПРОРЫВ В ЭКОНОМИКЕ", "ЛУЧШЕЕ ДЕТЯМ", "ЛУЧШЕЕ В ЭКОНОМИКЕ")*

*.flatMap(sentence -> Arrays.stream(sentence.split(" ")))*

*.forEach(System.out::println);*

* Операция ***limit()*** ограничивает стрим указанным количеством элементов.

*Stream.of("Пирамиды", "Эверест", "Колизей")*

*.limit(2)*

*.forEach(System.out::println);*

* Операция ***skip()*** отбрасывает указанное количество первых элементов стрима.

*Stream.of("Пирамиды", "Эверест", "Колизей")*

*.skip(2)*

*.forEach(System.out::println);*

* Операция ***distinct()*** позволяет получить стрим из уникальных элементов исходного стрима.

*Stream<Integer> names = Stream.of("Надя", "Ольга", "Марат", "Света", "Ева")*

*.map(String::length)*

*.distinct();*

*names.forEach(System.out::println);*

* Операция ***peek()*** позволяет выполнить дополнительное действие над элементами стрима без влияния на тип стрима.

*Stream.of(1, 1, 2, 2, 3, 3)*

*.peek(i -> System.out.println(“peek: “ + i)*

*.distinct()*

*.forEach(System.out::println);*

* Операция ***sorted()*** накапливает в себе элементы стрима, сортирует их и отправляет дальше.

*new Random().ints()*

*.limit(10)*

*.sorted()*

*.forEach(System.out::println);*

* Операции ***takeWhile()*** и ***dropWhile()*** с условиями (аналог циклов while). В момент, когда получает первый *false*, перестает работать (выходит из цикла) и пропускает все остальные элементы.

*Stream.of(1, 1, 2, 2, 3, 3)*

*.takeWhile(i -> i < 2)*

*.forEach(System.out::println);*

* Операция ***mapToObj()*** позволяет выполнить преобразование элемента стрима и запаковать его в объект.

*String str = “Hello world!”;*

*str.chars()*

*.mapToObj(c -> (char) c)*

*.forEach(System.out::println);*

Промежуточные операции выполняются только при вызове терминальной операции. Такое действие называется **ленивостью** (***laziness***).

В стримах также есть аналог цикла *for*:

*Stream.iterate(0, v -> v < 10, v -> v + 1) // Вывод чисел от 0 до 9*

*.forEach(System.out::println);*

У структур данных (коллекции, списки, массивы и т.д.) есть свой метод преобразования в стрим.

Любой момент времени в *Unix* представлен в виде целого числа ***timestamp*** – количестве миллисекунд от точки начала отсчета. Точка начала отсчета *Unix*-времени – 0 часов 0 минут 0 секунд 1 января 1970 года.

Класс ***Instant*** представляет метки времени, который хранит не только миллисекунды, но и наносекунд от начала отсчета. Экземпляры класса ***Instant*** являются неизменяемыми. Он реализует следующие методы:

* Статический метод ***now()*** возвращает текущую метку времени в виде экземпляра класса *Instant*.
* Метод ***toString()*** возвращает время в формате строки ISO-8601 вида 2021-11-23T12:45:50.123456Z
* Метод ***toEpochMilli()*** возвращает количество миллисекунд с начала Unix-эпохи до конкретной метки времени.
* Статические методы ***ofEpochSecond(long)*** и ***ofEpochMilli(long)*** позволяют создать метку времени, зная количество секунд или миллисекунд от момента события до *Unix*-эпохи. Статический метод ***ofEpochSecond(long second, long nano)*** позволяет создать такую метку с точностью до наносекунд.
* Статические методы ***MIN*** и ***MAX*** выводят границы допустимых к хранению дат.
* Методы ***plusSeconds(long)*** и ***minusSeconds(long)*** возвращают экземпляр ***Instant***, отличающийся на указанное количество секунд.
* Методы ***plusMillis(long)*** и ***minusMillis(long)*** возвращают экземпляр ***Instant***, отличающийся на указанное количество миллисекунд.
* Методы ***plusNanos(long)*** и ***minusNanos(long)*** возвращают экземпляр ***Instant***, отличающийся на указанное количество наносекунд.
* Методы ***isAfter(Instant otherInstant),*** ***isBefore(Instant otherInstant)*** и ***equals(Object otherInstant)*** возвращают булево значение при сравнении дат.

Класс ***LocalDateTime*** хранит текущую дату и время, полученную на текущем устройстве с помощью статического метода ***now()*** или ***of(год, месяц, день, часы, минуты, секунды, наносекунды)***. Экземпляры класса ***LocalDateTime*** являются неизменяемыми. Аналогично классу *Instant* у *LocalDateTime* есть:

* Методы для получения новых экземпляров на основе уже имеющихся значений – ***plusYears(long)***, ***minusYear(long)*** и другие.
* Методы ***isAfter(LocalDateTime*** ***otherMoment),*** ***isBefore(LocalDateTime*** ***otherMoment)*** и ***equals(LocalDateTime*** ***otherMoment)*** возвращающие булево значение при сравнении дат.
* Метод ***toString()*** возвращает время в формате строки ISO-8601 вида 2021-11-23T12:45:50.123456Z

Класс ***DateTimeFormatter*** описывает нужный формат вывода с помощью метода ***ofPattern(String)***.

*LocalDateTime now = LocalDateTime.now();*

*DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd.MM.yyyy, HH:mm");*

*String formatDateTime = now.format(formatter);*

При помощи статического метода ***parse(String)*** из строки можно получить экземпляр *LocalDateTime*.

*DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd.MM.yyyy, HH:mm");*

*LocalDateTime anotherDateTime = LocalDateTime.parse("22.02.2022, 22:22", formatter);*

Классы ***LocalDate*** и ***LocalTime*** оперируют только датой или только временем. Они поддерживают все методы класса ***LocalDateTime*** и работают аналогично. Дополнительно ни имеют методы преобразования:

* Методы ***toLocalTime()*** и ***toLocalDate()*** позволяют извлечь экземпляры соответствующих классов из объекта *LocalDateTime*.
* Статический метод ***of(LocalDate, LocalTime)*** позволяет собрать экземпляр *LocalDateTime*.

Из *LocalDateTime*, *LocalDate* и *LocalTime* можно извлечь нужные единицы времени с помощью соответствующих методов ***getYear()***, ***getMonth()***, ***getMonthValue()***, ***getDayOfMonth()***, и т.д.

У *LocalDateTime* и *LocalDate* есть еще два полезных метода:

* Метод ***getDayOfYear()*** возвращает порядковый номер дня в году.
* Метод ***getDayOfWeek()*** возвращает день недели – константу *java.time.DayOfWeek*.

Класс ***Period*** используется для вычисления срока между двумя датами. У экземпляра класса есть три поля – продолжительность в годах ***getYears()***, в месяцах ***getMonths()***, в днях ***getDays()***. Поддерживаемые методы:

* Статический метод ***between(LocalDate start, LocalDate end)*** позволяет узнать продолжительность. Последний день *end* в расчете не учитывается.
* Методы ***ofDays(int)***, ***ofWeeks(int)***, ***ofMonths(int)***, ***ofYears(int)*** создают экземпляр класса *Period*.
* Метод ***toString()*** выводит продолжительность в формате *P[количество лет]Y[количество месяцев]M[количество дней]D*. Если какая-то составляющая равна нулю, то она опускается.

Класс ***Duration*** используется для вычисления срока между двумя моментами времени и хранит продолжительность времени с точностью до наносекунд. Поддерживаемые методы:

* Статический метод ***between(LocalDate start, LocalDate end)*** позволяет узнать продолжительность. Последний день *end* в расчете не учитывается.
* Методы ***ofSeconds(long)***, ***ofSeconds(long seconds, long nanoseconds)***, ***ofMinutes(long)***, ***ofHours(long)***, ***ofDays(long)*** создают экземпляр класса *Duration*.
* Метод ***toString()*** выводит продолжительность в формате *P[количество часов]H[количество минут]M[количество секунд]S*. Наносекунды переводятся в секунды. Если какая-то составляющая равна нулю, то она опускается.
* Методы ***getSeconds()*** и ***getNano()*** возвращают значения соответствующих полей.
* Методы ***toDays()***, ***toHouras()***, ***toMinutes()***, ***toMillis()***, ***toNanos()*** конвертируют и возвращают соответствующее целое число.
* Методы ***toHoursPart()***, ***toMinutesPart()***, ***toSecondsPart()***, ***toMillisPart()*** конвертируют и возвращают количество соответствующей единицы времени от неполной единицы. Например, 75 минут выведется как 15 минут (неполная часть часа).

Родителем для классов ***Period*** и ***Duration*** является класс ***TemporalAmount***, поэтому они поддерживают метод ***plus(TemporalAmount)***.

Класс ***ZonedDateTime*** отвечает за временные зоны. По сути, это класс *LocalDateTime* с добавлением временной зоны ***ZoneId***. Временная зона может быть:

* **С фиксированным смещением**, например, *UTC0*. В этом случае время в любой точке мира будет одинаковым (*UTC0*). Для создания экземпляра с фиксированным смещением необходимо воспользоваться статическим методом ***of(String)***, передав ему смещение в формате *“UTC[+/-]hh:mm”*.
* **С привязкой к конкретному региону**. В этом случае время будет изменяться в зависимости от исторических данных (например, переход на зимнее время). Для создания экземпляра с фиксированным смещением необходимо воспользоваться статическим методом ***of(String)***, передав ему смещение в формате *“Europe/Moscow”*.

С помощью этого метода можно собрать экземпляр *ZoneDateTime* из *LocalDateTime* и *ZoneId*. Также этот класс поддерживает методы *LocalDateTime* и *ofInstant(Instant, ZoneId)*.

Метод ***withZoneSameInstant(ZoneId)*** работает аналогично переводу часов – момент времени остается неизменным, меняется *LocalDateTime* и *ZoneId*.

Метод ***withZoneSameLocal(ZoneId)*** работает аналогично выбору другого часового пояса – время показывается то же самое, но зона будет другой.

Форматирование для *ZonedDateTime* идентично *LocalDateTime*, с той лишь разницей, что можно отобразить название и смещение временной зоны. Например, *"Время: HH:mm:ss. Регион: VV, смещение: ZZZZZ"*.

**Сервер** – компьютер, который предоставляет информацию другим компьютерам. Такое сообщение называется **ответом** (***response***).

**Клиент** – компьютер, который запрашивает информацию у сервера. Такое сообщение называется **запросом** (***request***). Сервер тоже может быть клиентом.

***Internet Protocol (IP)*** – регламентирует подготовку данных к передаче и их доставку адресату.

***Transmission Control Protocol (TCP)*** – регламентирует доставку и проверяет, что IP передал все данные адресату по определенному порту. Если нет, они запрашиваются и отправляются повторно.

**Сетевой порт (*port*)** – целое неотрицательное число, которое закрепляется за конкретной программой.

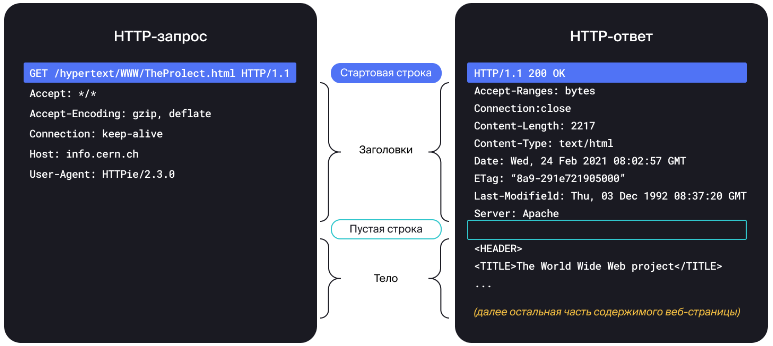
***HyperText Transfer Protocol (HTTP)*** – протокол передачи гипертекста использует протоколы *TCP* и *IP*, вводит понятия клиента и сервера и описывает правила их взаимодействия.

***Uniform Resource Locator (URL)*** – уникальный адрес на ресурс или гиперссылка.



***Domain Name System (DNS)*** – система доменных имен позволяет привязать понятное имя к *IP*-адресу и применяться вместо него.

***HyperText Markup Language (HTML)*** – язык гипертекстовой разметки используется для создания сайтов и позволяет оставлять в тексте метки (**теги**) в угловых скобках *< >*.



Каждое обращение к серверу начинается со стартовой строки. В ней указывается имя протокола и его методы взаимодействия с сервером. Основные методы, применяемые в *HTTP*-запросах:

* *POST* – используется для передачи данных на сервер.
* *GET* – используется для получения ресурса.
* *PUT* – используется для передачи данных серверу аналогично *POST*. Разница в том, что он заменяет существующий ресурс целиком или создает новый ресурс при его отсутствии.
* *PATCH* – используется для частичного обновления данных ресурса.
* *DELETE* – используется для удаления ресурса.

Методы соглашения ***CRUD*** (*Create*, *Read*, *Update*, *Delete*) сопоставляются соответственно с методами *POST*, *GET*, *PUT*, *DELETE*.

В ответе сервера на запрос **в стартовой строке** указывается **код ответа** (***status code***). Он состоит из трехзначного числа и названия. Коды успеха начинаются с 2, коды клиентских ошибок – с 4, коды серверных ошибок – с 5.

Дополнительная информация передается **в заголовке** (***headers***). Каждый заголовок состоит из названия и значения, отделенного двоеточием. Примеры имен заголовков:

* *Accept* – используется только в запросах и указывает серверу список типов данных, которые клиент может от него принять и корректно обработать. Значение *\*/\** означает любые типы данных.
* *Host* – используется только в запросах и указывает адрес сервера, которому нужно отправить запрос.
* *Content-Length* – указывает размер тела сообщения в байтах.
* *Content-Type* – указывает тип передаваемых данных.

Также вместе с запросом или ответом могут передаваться какие-либо данные. Они помещаются в **тело** (***body***) и отделяются пустой строкой.

**Веб-приложения** (***web applications***) – это сайты, состоящие из динамических страниц и представляющие возможность интерактивного взаимодействия с ними. **Фронтенд часть** веб-приложения решает, как отобразить данные в браузере, а также вовремя обновляет их. Серверы, обрабатывающие запросы фронтенда веб-приложения, называются **веб-сервисами**, или **веб-службами** (***web services***).

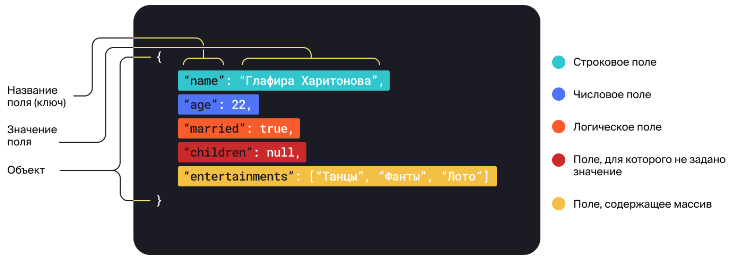
***Web Application Programming Interface*** *(****Web API****)* – это набор *URL* и совокупность соглашений о том, как именно работать с ними другим программам. Например, приложения могут взаимодействовать друг с другом через *API* по принципу черного ящика.

При разработке приложений могут использоваться подходы ***API-first*** или ***Code-first***.



***JavaScript Object Notation (JSON)*** – специальный формат для передачи данных. Во фронтенде он используется в *POST*, *PUT* и *PATCH* запросах. В бэкенде он используется в *GET* и *PUT* запросах.

Формат *JSON* поддерживает шесть типов данных. Примитивы – строки, числа, логические значения, *null*. Сложные типы – массивы *[1,2,3,4]* и объекты *{ “value”: 42 }*. Примитивы не могут использоваться сами по себе, они должны входить в состав объекта или массива. Есть поддержка вложенных типов.



**Рассмотрим создание *HTTP*-сервера.**

**Эндпоинт (*end-point*)**, или конечная точка – это комбинация *URL*-адреса и *HTTP*-метода. Каждый эндпоинт отвечает за выполнение конкретной задачи.

В стандартной библиотеке *Java* есть свой *HTTP*-сервер. Он находится в пакете *com.sun.net.httpserver* и реализован главным классом ***HttpServer***.

При передаче по сети информация разбивается на блоки – **сетевые пакеты** (***network packet***). Внутри пакетов есть данные, а также *IP*-адрес и **номер порта**, используемого для передачи данных. Номер порта привязан к определенной программе и принимает значения от 1 до 65535 (216-1), а также уникален в данной операционной системе.

**Сокет (*socket*)** – это средство операционной системы, через которое программа может получить доступ к сети. В качестве параметров для открытия сокета используются *IP*-адрес и номер порта. Одной программе могут соответствовать несколько сокетов.

Метод ***HttpServer.create()*** позволяет создать собственный веб-сервер.

Метод ***HttpServer.bind(InetSocketAddress addr, int backlog)*** привязывает сервер к порту.

Числовой параметр ***backlog*** – это число соединений, которые могут находиться в ожидании, пока сервер обрабатывает текущий запрос. При указании нуля будет использоваться стандартное значение, определенное в операционной системе. Если не указывать конкретный *IP*-адрес, то веб-сервер будет работать на всех доступных *IP*-адресах.

Процесс трансформации Java-объекта в какой-то другой формат называется **сериализацией** (***serialization***). Обратный процесс – **десериализация** (***deserialization***). Популярные форматы для сериализации данных – *JSON*, *XML*, *Protobuf*, *YAML*.

Рассмотрим пример такой трансформации в *JSON* с помощью библиотеки ***GSON***. Метод ***toJson(Object src)*** переводит объект в строковое представление *JSON*. Метод ***fromJson(String json, Class<T> classOfT)*** переводит строковое представление *JSON* в указанный объект класса.

*import com.google.gson.Gson;*

*class Practicum {*

*public static void main(String[] args) {*

*Gson gson = new Gson();*

*String jsonString = gson.toJson(dog); // сериализация*

*Dog dog = gson.fromJson(jsonString, Dog.class); } } // десериализация*

Для создания экземпляра *Gson* с дополнительными настройками используется класс ***GsonBuilder***.

Метод ***create()*** возвращает объект класса *Gson*.

Метод ***setPrettyPrinting()*** включает удобное для чтения отображение данных в формате *json*. По умолчанию эта настройка выключена и *json* выводится в одну строку.

Метод ***serializeNulls()*** включает добавление в json строк со значением null. По умолчанию эта настройка выключена и *json* не добавляет поля вида *"название\_поля": null*.

Метод ***registerTypeAdapter(Type type, Object typeAdapter)*** позволяет реализовать свой сериализатор или десериализатор с заданными параметрами. Для этого нужно написать свой конвертер, унаследованный от абстрактного класса ***TypeAdapter***, и переопределить методы ***write(JsonWriter jsonWriter, Object obj)*** и ***read(JsonReader jsonReader, Object obj)***.

Например, переопределим формат конвертирования для экземпляра даты *LocalDate*:

*class LocalDateAdapter extends TypeAdapter<LocalDate> {*

*private static final DateTimeFormatter formatterWriter = DateTimeFormatter.ofPattern("dd--MM--yyyy");*

*private static final DateTimeFormatter formatterReader = DateTimeFormatter.ofPattern("dd.MM.yyyy");*

*@Override*

*public void write(final JsonWriter jsonWriter, final LocalDate localDate) throws IOException {*

*jsonWriter.value(localDate.format(formatterWriter)); }*

*@Override*

*public LocalDate read(final JsonReader jsonReader) throws IOException {*

*return LocalDate.parse(jsonReader.nextString(), formatterReader); } }*

*Gson gson = new GsonBuilder()*

*.registerTypeAdapter(LocalDate.class, new LocalDateAdapter())*

*.create();*

Для создания эндпоинта нужно создать класс, реализующий интерфейс ***HttpHandler*** и переопределить метод ***handle(HttpExchange httpExchange)***.

Метод ***httpExchange.createContext(String path, HttpHandler handler)*** используется, чтобы связать *URL* путь и обработчик запроса.

Метод ***httpExchange.start()*** используется для запуска сервера.

Метод ***httpExchange.sendResponseHeaders (int rCode, long responseLength)*** отправляет ответ, который можно сформировать заранее.

Если в ответ нужно включить тело, то требуется создать экземпляр класса ***OutputStream*** с помощью метода ***httpExchange.getResponseBody()***. Он возвращает объект ***OutputStream***, в который нужно записать поток байтов с помощью метода ***getBytes()***.

В итоге базовый сервер будет выглядеть так:

*public class Practicum {*

*private static final int PORT = 8080;*

*public static void main(String[] args) throws IOException {*

*HttpServer httpServer = HttpServer.create(); // создаем сервер*

*httpServer.bind(new InetSocketAddress(PORT), 0); // назначаем сокет*

*httpServer.createContext("/hello", new HelloHandler()); // связываем путь и обработчик*

*httpServer.start(); // запускаем сервер*

*System.out.println("HTTP-сервер запущен на " + PORT + " порту!"); }*

*static class HelloHandler implements HttpHandler {*

*@Override*

*public void handle(HttpExchange httpExchange) {*

*try {*

*String response = "Hey! Glad to see you on our server.";*

*httpExchange.sendResponseHeaders(200, 0);*

*try (OutputStream os = httpExchange.getResponseBody()) {*

*os.write(response.getBytes()); } } // преобразование тела в байты*

*catch(Exception e) {*

*e.printStackTrace(); }*

*finally {*

*httpExchange.close(); } // закрытие потока*

Метод ***httpExchange.getRequestBody()*** возвращает тело запроса в формате ***InputStream***, которые клиент отправил на сервер.

*InputStream inputStream = httpExchange.getRequestBody();*

*String body = new String(inputStream.readAllBytes(), StandardCharsets.UTF\_8);*

Метод ***httpExchange.getRequestMethod()*** возвращает *HTTP*-метод, который клиент использовал при отправке запроса. С помощью него можно сделать из одного эндпоинта несколько.

*String method = httpExchange.getRequestMethod();*

*String response;*

*switch(method) {*

*case "POST":*

*response = "Вы использовали метод POST!";*

*break;*

*case "GET":*

*response = "Вы использовали метод GET!";*

*break;*

*default:*

*response = "Вы использовали какой-то другой метод!"; }*

Метод ***httpExchange.getRequestHeaders()*** возвращает заголовки, которые передал клиент, в виде *Map<String, List<String>>*.

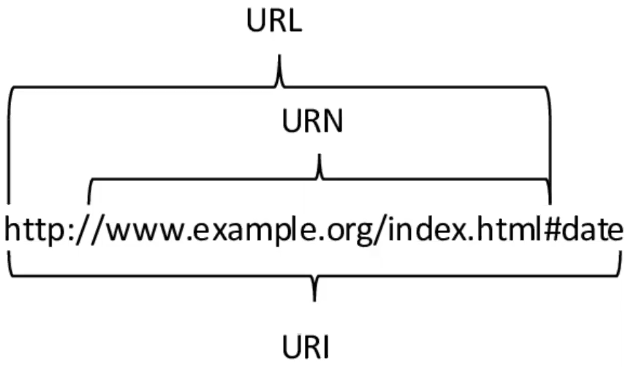
Метод ***httpExchange.getRequestURI()*** возвращает *URI*, использованный клиентом при вызове сервера. Из него можно получить данные пути с помощью метода ***getPath()*** и использовать в ответе.

Метод ***httpExchange.getResponseHeaders()*** возвращает объект класса ***Headers***, в который можно добавить заголовки с помощью метода ***set(String)***.

**Рассмотрим создание *HTTP*-клиента.**

Для этого нам понадобятся:

* Класс ***java.net.URI*** – описывает адрес интересующего ресурса. Его можно создать с помощью статического метода ***create(String)***.



* Класс ***HttpRequest*** – описывает *HTTP*-запрос к серверу. Статический метод ***newBuilder().build()*** возвращает запрос с настройками по умолчанию.
* Класс ***HttpResponse*** – описывает *HTTP*-ответ к серверу. Методы ***body()*** и ***statusCode()*** позволяют получить соответствующие строковые значения.
* Класс ***HttpClient*** – реализует коммуникацию с сервером по протоколу *HTTP*. Статический метод ***newHttpClient()*** возвращает клиент с настройками по умолчанию. Статический метод ***send(HttpRequest request, HttpResponse.BodyHandler<T> responseBodyHandler)*** отправляет запрос на сервер и возвращает объект типа *HttpResponse*.

Пример простого *HTTP*-клиента:

*public class Practicum {*

*public static void main(String[] args) throws IOException, InterruptedException {*

*// создаём экземпляр URI, содержащий адрес нужного ресурса*

*URI url = URI.create("https://ru.wikipedia.org/wiki/Список\_кодов\_состояния\_HTTP");*

*// создаём объект, описывающий HTTP-запрос*

*HttpRequest request = HttpRequest.newBuilder() // получаем экземпляр билдера*

*.GET() // указываем HTTP-метод запроса*

*.uri(url) // указываем адрес ресурса*

*.version(HttpClient.Version.HTTP\_1\_1) // указываем версию протокола*

*.header("Accept", "text/html") // указываем заголовок Accept*

*.build(); // заканчиваем настройку и создаём http-запрос*

*// HTTP-клиент с настройками по умолчанию*

*HttpClient client = HttpClient.newHttpClient();*

*// получаем стандартный обработчик тела запроса с конвертацией содержимого в строку*

*HttpResponse.BodyHandler<String> handler = HttpResponse.BodyHandlers.ofString();*

*// отправляем запрос и получаем ответ от сервера*

*HttpResponse<String> response = client.send(request, handler);*

*// выводим код состояния и тело ответа*

*System.out.println("Код ответа: " + response.statusCode()); // Можно добавить реализацию в*

*System.out.println("Тело ответа: " + response.body()); } } // зависимости от кода ответа*

Исключения ***ConnectException*** и другие, наследуемые от ***IOException***, возникают из-за проблем с сетевой коммуникацией (перегружен сервер, обрыв соединения, несуществующий адрес, нет связи с сервером).

Исключения ***InterruptedException*** возникает при долгом ожидании ответа от сервера, когда запрос обрабатывается на стороне клиента.

Для передачи данных на сервер используется не только тело запроса, но и **строка запроса** (***query string***). Она находится в конце *URL* и отделяется от пути знаком вопроса **?** , после которого указываются пары в формате ***параметр=значение***, отделенные символом **&** .

Адрес может содержать только латинские буквы, цифры и отдельные знаки пунктуации. Все остальные символы будут преобразованы через **процентное кодирование**.



Также для передачи данных на сервер можно использовать заголовки (***header***). Для этого нужно вызвать метод ***header(String name, String value)*** у билдера в момент формирования запроса. Например, *header("Accept", "application/*json*")* в запросе информирует сервер, что клиент готов получить ответ в формате *json*.

Для работы с форматом GSON существует несколько полезных библиотек:

*import com.google.gson.JsonArray; // описывает JSON-массив*

*import com.google.gson.JsonElement; // описывает любой тип данных JSON*

*import com.google.gson.JsonObject; // описывает JSON-объект*

*import com.google.gson.JsonParser; // разбирает JSON-документ на элементы*

Метод ***JsonParser.parseString(String json)*** разбирает текст на составные элементы и возвращает объект класса ***JsonElement***.

Метод ***JsonObject.get(String name)*** принимает название поля и возвращает его значение в виде ***JsonElement***.

Методы класса ***JsonElement***:

* ***isJsonObject()*** – проверяет, является ли элемент *JSON*-объектом
* ***getAsJsonObject()*** – преобразовывает элемент в *JSON*-объект
* ***getAsJsonArray()*** – преобразовывает элемент в *JSON*-массив
* ***getAsString()*** – преобразовывает *JSON*-элемент в строку
* ***getAsLong()*** – преобразовывает *JSON*-элемент в число
* ***getAsBoolean()*** – преобразовывает *JSON*-элемент в логическое значение

*if (response.statusCode() == 200) {*

*// передаем парсеру тело ответа в виде строки, содержащей данные в формате JSON*

*JsonElement jsonElement = JsonParser.parseString(response.body());*

*if(!jsonElement.isJsonObject()) { // проверяем, точно ли мы получили JSON-объект*

*System.out.println("Ответ от сервера не соответствует ожидаемому.");*

*return; }*

*// преобразуем результат разбора текста в JSON-объект*

*JsonObject jsonObject = jsonElement.getAsJsonObject();*

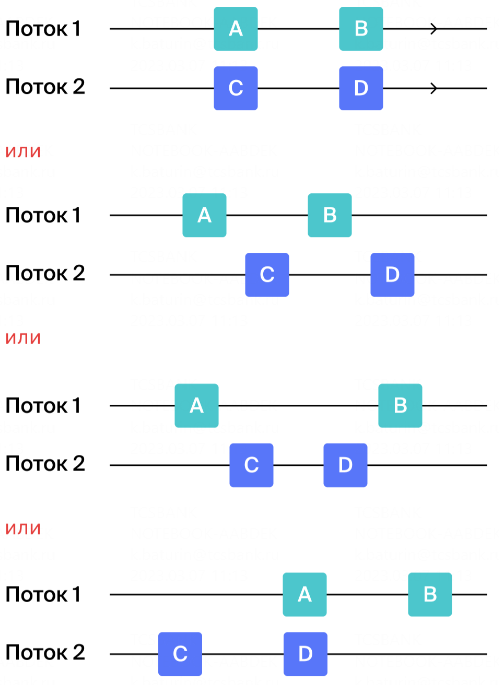
*// получаем название страны*

*String country = jsonObject.get("country").getAsString();*

*System.out.println("Страна: " + country); }*

**Поток** – это инструмент, который позволяет выполнять части программы одновременно. Потоки представлены классом ***Thread***.

В рамках одного потока, операции выполняются последовательно. Между двумя потоками операции могут выполняться в любом порядке.



Поток может быть:

* **выполняющимся**
* **готовым к выполнению**, как только он получит время и ресурсы ЦП
* **приостановленным**, т.е. временно не выполняющимся
* **возобновленным** в дальнейшем
* **заблокированным** в ожидании ресурсов для своего выполнения
* **завершенным**, когда его выполнение закончено и не может быть возобновлено

Создать поток можно двумя способами: либо с помощью реализации интерфейса *Runnable*, либо путем создания подкласса класса *Thread*. Один и тот же объект типа *Runnable* можно передавать нескольким потокам.

*Thread thread = new Thread(() -> {* набор действий *}); // с помощью лямбды*

*Runnable runnable = () -> {* набор действий *}; // явно создав Runnable*

*Thread thread = new Thread(runnable);*

*Runnable runnable = () -> { // создав анонимный класс*

*@Override*

*Public void run() {* набор действий *} };*

*Thread thread = new Thread(runnable);*

*Thread thread = new Thread() { // унаследовавшись от класса Thread и переопределив метод run*

*@Override*

*public void run() {* набор действий *} };*

*public class MyLogic implements Runnable { // создав класс реализующий Runnable*

*@Override*

*public void run() {* набор действий *} };*

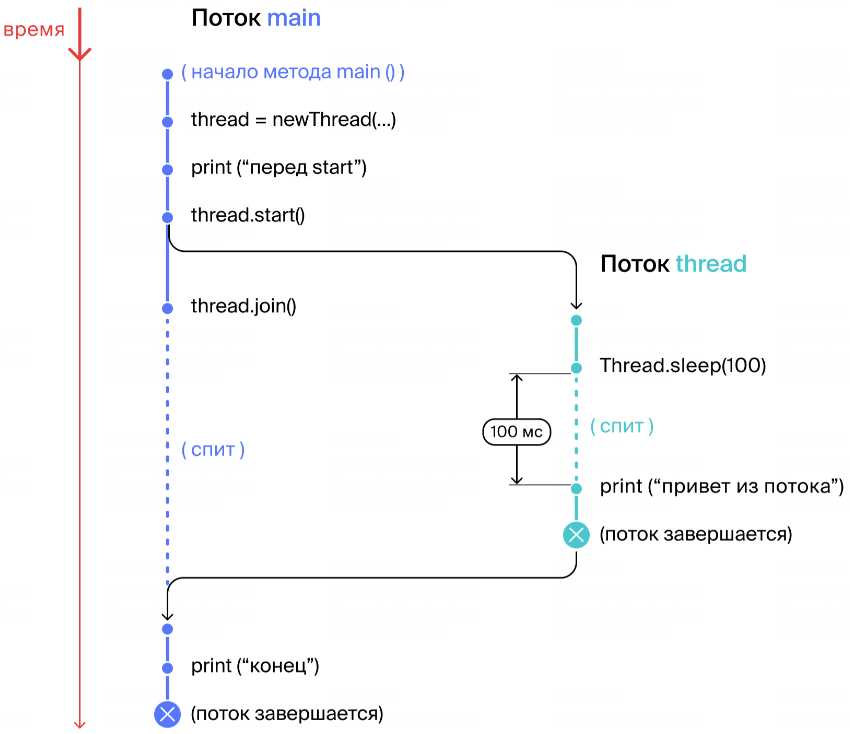
*Thread thread = new Thread(new Mylogic());*

В каждом процессе существует ***main* (основной)** поток. Все остальные подчиненные потоки могут быть порождены им. Программа завершается, когда завершаются все её потоки.

Метод ***thread.start()*** запускает выполнение потока.

Метод ***thread.sleep(100L)*** приостанавливает выполнение потока на 100 милисекунд.

Метод ***thread.join()*** блокирует выполнение потока, из которого был вызван, до завершения указанного потока или ждет указанное время.



*public class EndOfMain {*

*public static void main(String[] args) throws InterruptedException {*

*Thread thread = new Thread(() -> {*

*try { Thread.sleep(100L);*

*} catch (InterruptedException e) {*

*Thread.currentThread().interrupt();*

*return; }*

*System.out.println("Привет из потока"); }); // вторая строка*

*System.out.println("main(): перед thread.start()"); // первая строка*

*thread.start();*

*thread.join();*

*System.out.println("main(): конец"); } } // третья строка*

Метод ***thread.interrupt()*** прерывает выполнение любого блокирующего метода в потоке (например, *sleep()*) и заставляет прерванный метод выбросить исключение ***InterruptedException***. По факту этот метод устанавливает флаг *interrupt* в положение *true*. Блокирующие методы видят этот флаг, прекращают блокировку и устанавливают флаг в положение *false*. После этого поток доходит до логического завершения. Для корректного использования цикла *while* рекомендуется следующая конструкция:

*Thread thread = new Thread(() -> {*

*while (!Thread.currentThread().isInterrupted()) {*

*System.out.println("SPAM"); } });*

Метод ***thread.stop()*** принудительно прерывает выполнение потока. Не рекомендован к использованию, т.к. могут повредиться данные.

С точки зрения процессора операции инкремента/декремента – это три атомарные операции (прочитать, изменить, записать), которые могут перемешиваться в режиме многопоточности. Для работы с многопоточностью используется популярный пакет ***java.util.concurrent***.

Этот пакет содержит классы ***AtomicInteger***, ***AtomicLong***, ***AtomicReference*** и другие классы для корректной работы в режиме многопоточности. Конкретно для инкремента/декремента используются методы ***AtomicInteger.getAndIncremet()*** и ***AtomicInteger.getAndDecremet()***.

Сделать операцию атомарной можно с помощью оператора ***synchronized***. Этот оператор защищает куски кода от одновременного выполнения разными потоками, создавая **однопоточную критическую секцию**. Критические секции могут быть сгруппированы с одним и тем же объектом-монитором. Он не может быть примитивом и гарантирует последовательность выполнения секций с одним и тем же объектом-монитором. Объявить метод синхронизированным можно несколькими способами:

* объявить при описании метода:

*synchronized int sumArray(int nums[]) { … }*

* вызвать метода внутри блока кода, объявленного как синхронизированный:

*synchronized(ссылка\_на\_объект-монитор) { // синхронизируемые инструкции }*

*Synchronized* может значительно замедлить выполнение программы в отличии от *Atomic*, но является универсальным средством.

*public class IncDecThreadsSynchronized {*

*private static final int N = 100\_000\_000;*

*private static int x = 0;*

*public static void main(String[] args) throws InterruptedException {*

*Object monitor = new Object();*

*Thread incThread = new Thread(() -> {*

*for (int i = 0; i < N; i++) {*

*synchronized (monitor) {*

*x++; } } });*

*Thread decThread = new Thread(() -> {*

*for (int i = 0; i < N; i++) {*

*synchronized (monitor) {*

*x--; } } });*

*incThread.start();*

*decThread.start();*

*incThread.join();*

*decThread.join();*

*System.out.println("x = " + x); } } // в результате всегда выведет x = 0*

Обычные реализации коллекций не поддерживают работу с многопоточностью. Вместо них нужно использовать специальные версии из пакета *java.util.concurrent*. Например, ***ConcurrentSkipListSet***, ***ConcurrentHashMap*** и другие.

*public class TwoThreadsAndSet {*

*private static final int N = 100\_000;*

*public static void main(String[] args) throws InterruptedException {*

*// отличие от предыдущей программы в этой строке:*

*Set<Integer> set = new ConcurrentSkipListSet<>();*

*for (int i = 0; i < N; i++) {*

*set.add(i); }*

*Thread oddThread = new Thread(() -> {*

*for (int i = 0; i < N; i++) {*

*if (i % 2 == 1) {*

*set.remove(i); } } });*

*Thread evenThread = new Thread(() -> {*

*for (int i = 0; i < N; i++) {*

*if (i % 2 == 0) {*

*set.remove(i); } } });*

*oddThread.start();*

*evenThread.start();*

*oddThread.join();*

*evenThread.join();*

*System.out.printf("set.size() = %d%n", set.size()); } } // в результате всегда выведет set.size() = 0*

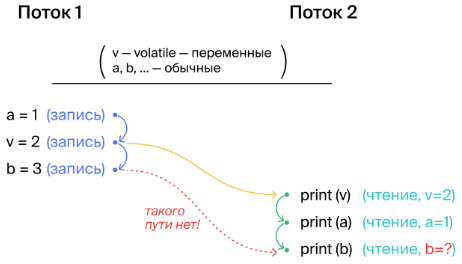
**Взаимная блокировка потоков (*deadlock*)** – это одна из частых причин зависания программ, когда разные потоки вечно ждут друг друга. Другие причины – это бесконечные циклы, нехватка памяти и медленные операции ввода-вывода.

Чтобы избежать взаимных блокировок, можно договориться в каком порядке все потоки производят захваты, или не использовать общие объекты-мониторы.

По умолчанию *Java* не гарантирует, что значение, присвоенное в одном потоке, будет прочитано в другом потоке. Для исправления этой проблемы используется ключевое слово ***volatile*** при объявлении переменной. Оно сообщает *Java*, что эта переменная используется несколькими потоками и все потоки должны видеть актуальное значение переменной.

**Правило видимости значений** в упрощенном виде:

* Значения, записанные в рамках одного потока, видны последующим операциям в том же потоке.
* *volatile*-переменные добавляют стрелку от операции записи к последующим операциям чтения, даже если операции выполнены в разных потоках.
* Если от операции записи до операции чтения можно дойти стрелками любого цвета, то операция чтения гарантировано получит записанное значение (*print(a)*).
* Если по стрелкам пройти нельзя, то получение записанного значения не гарантировано.



Если в приложении нет общих изменяемых переменных, то нет и связанных с ними проблем – области видимости и гонки потоков.

**Пул** – это ограниченный набор ресурсов, которые можно использовать многократно. Для работы с пулом потоков в *Java* используется интерфейс ***ExecutorService***. Он сам подбирает свободный поток из пула и передает ему задачу на выполнение. Задачи представлены интерфейсов *Runnable*, как при создании потоков. *ExecutorService* можно создать с помощью статических методов класса ***Executors***.

К задачам, выполняющимся в пуле потоков, применимы те же правила, что и к задачам, запущенным с использованием *Thread* (гонка данных, области видимости и т.д.).

Внутри пула потоков все добавленные задачи сохраняются в очередь и берутся в той же последовательности (первый зашел – первый вышел), но могут выполняться в другом порядке.

*public class ThreadPoolExample {*

*public static void main(String[] args) throws InterruptedException {*

*List<Runnable> tasks = new ArrayList<>();*

*for (int i = 1; i <= 100\_000; i++) {*

*final int copyOfI = i; // нужно для лямбда-выражения*

*Runnable task = () -> {*

*System.out.println("Задача #" + copyOfI); };*

*tasks.add(task); }*

*ExecutorService executor = Executors.newFixedThreadPool(10); // создаем пул из 10 потоков*

*for (Runnable task : tasks) {*

*executor.execute(task); }*

*executor.shutdown();*

*while (!executor.awaitTermination(1, TimeUnit.SECONDS)) { // ежесекундный цикл проверки*

*/\* пусто \*/ } } }*

Для передачи данных между потоками могут использоваться интерфейс многопоточной очереди ***BlockingQueue*** из пакета *java.util.concurrent*. Если очередь переполнена, то ***put()*** будет ждать, пока в очереди появится место. Есть очередь пустая, то ***take()*** будет ждать, пока в очередь что-нибудь добавят. Эти методы блокирующие друг друга.

*public class ProducerAConsumerB {*

*private static final int N = 4;*

*private static final int QUEUE\_SIZE = 2;*

*private static final int POISON\_PILL = -1;*

*public static void main(String[] args) throws InterruptedException {*

*BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(QUEUE\_SIZE);*

*Thread threadA = new Thread(() -> {*

*try {*

*for (int i = 1; i <= N; i++) {*

*queue.put(i);*

*System.out.println("A: добавил " + i + " в очередь"); }*

*queue.put(POISON\_PILL); // объект-флаг о завершении передачи*

*System.out.println("A: добавил " + POISON\_PILL);*

*} catch (InterruptedException e) {*

*Thread.currentThread().interrupt(); } });*

*Thread threadB = new Thread(() -> {*

*try {*

*while (true) {*

*int i = queue.take();*

*System.out.println("B: получил " + i);*

*if (i == POISON\_PILL) {*

*System.out.println("B: конец");*

*break; } }*

*} catch (InterruptedException e) {*

*Thread.currentThread().interrupt(); } });*

*threadA.start();*

*threadB.start();*

*threadA.join();*

*threadB.join();*

*System.out.println("main(): конец"); } }*

Более подробно почитать про многопоточность можно [тут](https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html).

Для координации синхронизированных потоков между собой используются методы ***wait()***, ***notify()*** и ***notifyAll()***. Метод *wait()* переводит поток в режим ожидания до поступления уведомлений или на указанный промежуток времени (рекомендовано использовать в цикле с условием). Методы *notify()* и *notifyAll()* возобновляет выполнение одного или всех ожидающих потоков.

Механизмы приостановки, возобновления и остановки потоков реализуются соответствующими методами класса Thread – ***suspend()***, ***resume()*** и ***stop()***. Однако пользоваться ими напрямую не рекомендуется в связи с возможностью взаимоблокировки всех потоков. Лучше использовать эти методы через логические переменные-флаги.

**Модуль** – совокупность пакетов и ресурсов, доступных по имени модуля. Начиная с JDK 9 модули доступны для *Java API*. Компиляция модульных программ происходит с помощью ***javac***. Модульное приложение может быть упаковано в файл *JAR* (*Java Archive*).

*module имя\_модуля {*

*// определение модуля }*

Также можно определить по умолчанию открытый модуль с помощью ключевого слова ***open module***. Это позволяет сделать доступными пакеты модуля для рефлексии. **Рефлексия** – механизм, позволяющий программе анализировать код на этапе выполнения.

*open module имя\_модуля {*

*// определение модуля }*

Для открытия доступа только к определенному пакету модуля используется ключевое слово ***opens***.

*opens имя\_пакета;*

Для объявления зависимости одного модуля от другого используется ключевое слово ***requires***. Для объявления доступности пакетов модуля для других используется ключевое слово ***exports***. Для сужения открываемого доступа к модулю используется ключевое слово ***to*** (**квалифицированный экспорт**). Для предоставления возможности неявного чтения (A зависит от B, B зависит от С, A хочет читать C) используется ключевое слово ***transitive***. Ключевое слово ***static*** позволяет указать, что модуль требуется только для компиляции, но не для выполнения.

*requires имя\_модуля;*

*requires transitive имя\_модуля;*

*requires static имя\_модуля;*

*exports имя\_пакета;*

*exports имя\_пакета to имена\_модулей;*

В силу своей значимости модуль *java.base* автоматически доступен для других модулей. Код, который не является частью именованного модуля, автоматически становится частью **безымянного модуля**. В безымянном модуле все пакеты автоматически экспортируются, и он имеет доступ к любым другим модулям, а вместо пути модуля используется путь класса при компиляции.

В Java поддерживается архитектура подключаемых приложений благодаря использованию **служб** и **провайдеров служб**. **Служба** – это программная единица, функциональность которой определяется интерфейсом или абстрактным классом. **Провайдер** предоставляет конкретную реализацию службы. Например, служба определяет возможность перевода на другой язык, а разные провайдеры реализуют перевод на нужный язык.

*class ServiceLoader<S> // класс определяющий провайдеры*

*public static <S> ServiceLoader<S> load(Class <S> тип\_службы) // загрузка провайдера*

Ключевой слово ***provides*** используется для указания, что модуль реализует службу. Ключевой слово ***uses*** используется для указания службы, которая требуется модулю. Ключевой слово ***with*** используется для указания конкретного типа провайдера службы.

*provides тип\_службы with типы\_реализации;*

*uses тип\_службы;*