**Yandex Practicum + Герберт Шилдт (полное рук-во) + HSE COURSE + CHAT GPT PRO**

**JAVA ARHITECTURE**

Надежность программ обсулавливается отсутствием адресной арифметики, проверки на этапе

Кросплатформенность программ достигается с помощью того, что компилятор javac единый для всех процессоров, но сам интерпретатор промежуточного байт кода JVM пишется отдельно под каждую архитектуру. Именно поэтому SDK не кроссплатформенный, так как содержит JVM.

Изображение выглядит как текст, рукописный текст, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Для запуска программы:

1. Переходим в директорию с main методом – cd src
2. Компилируем главный класс - javac Main.java
3. Запускаем получившийся файл .class – java Main

Либо в одну команду – java Main.java

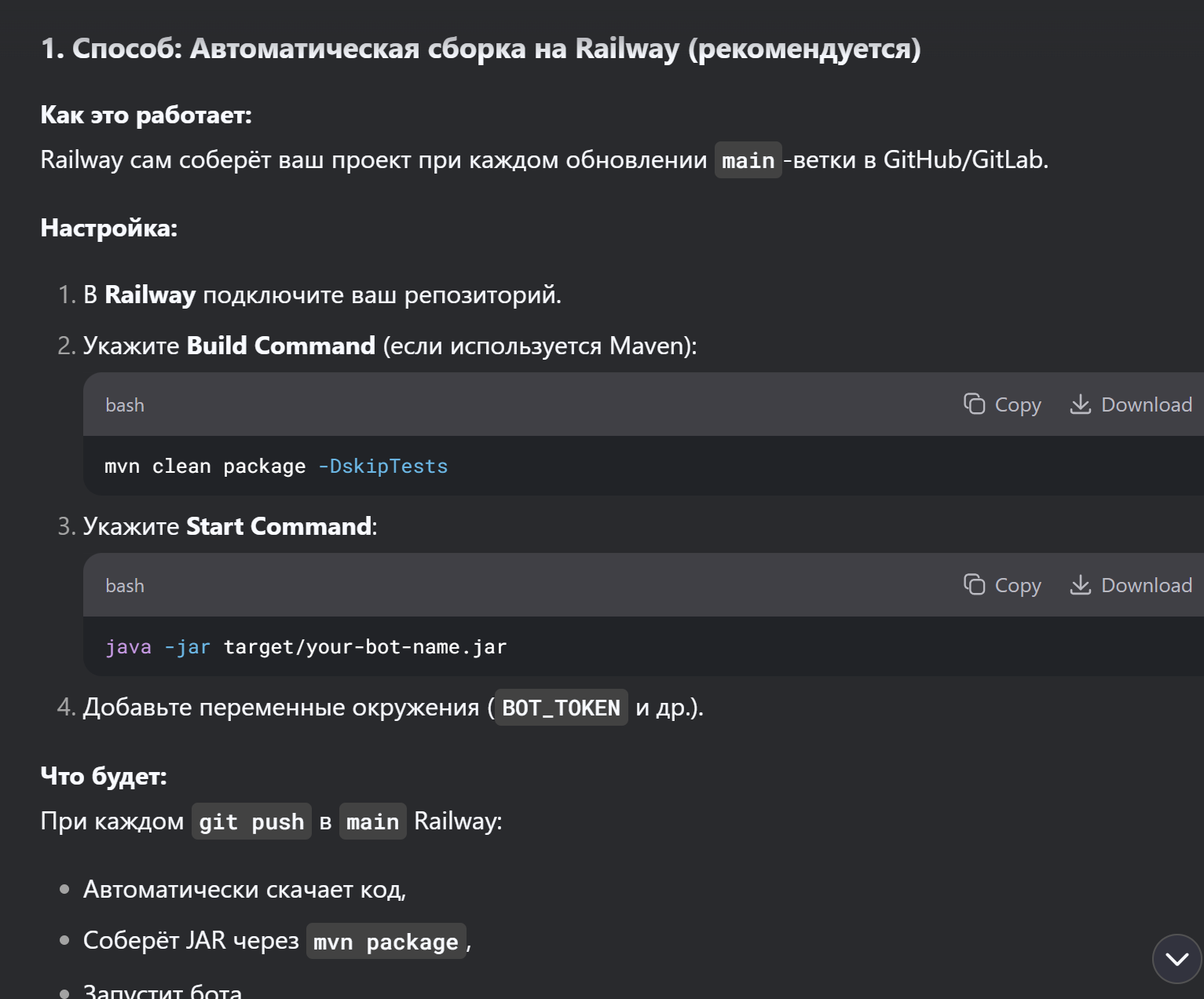
**CI/CD**

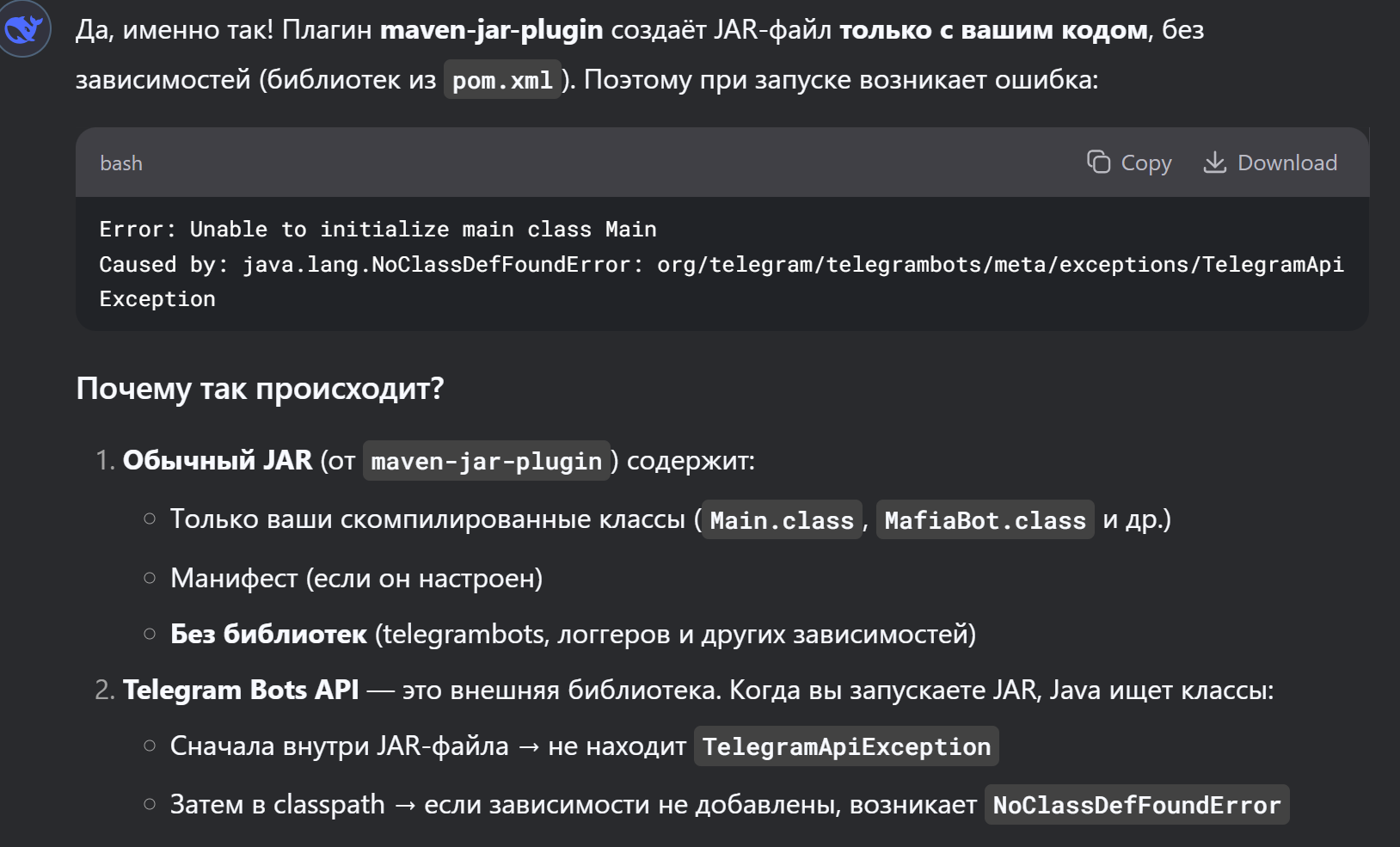
Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

IDEA использует свой локальный Maven, поэтому на ПК он не установлен и команды в терминале с mvn недоступны. Нужно скачать именно БИНАРНЫЙ архив Maven, тк обычный src содержит .java файлы для разработки самого Maven, а в бинарном – исполняемые файлы 



Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

**COMMIMTING CONVENTION**Изображение выглядит как текст, снимок экрана, программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

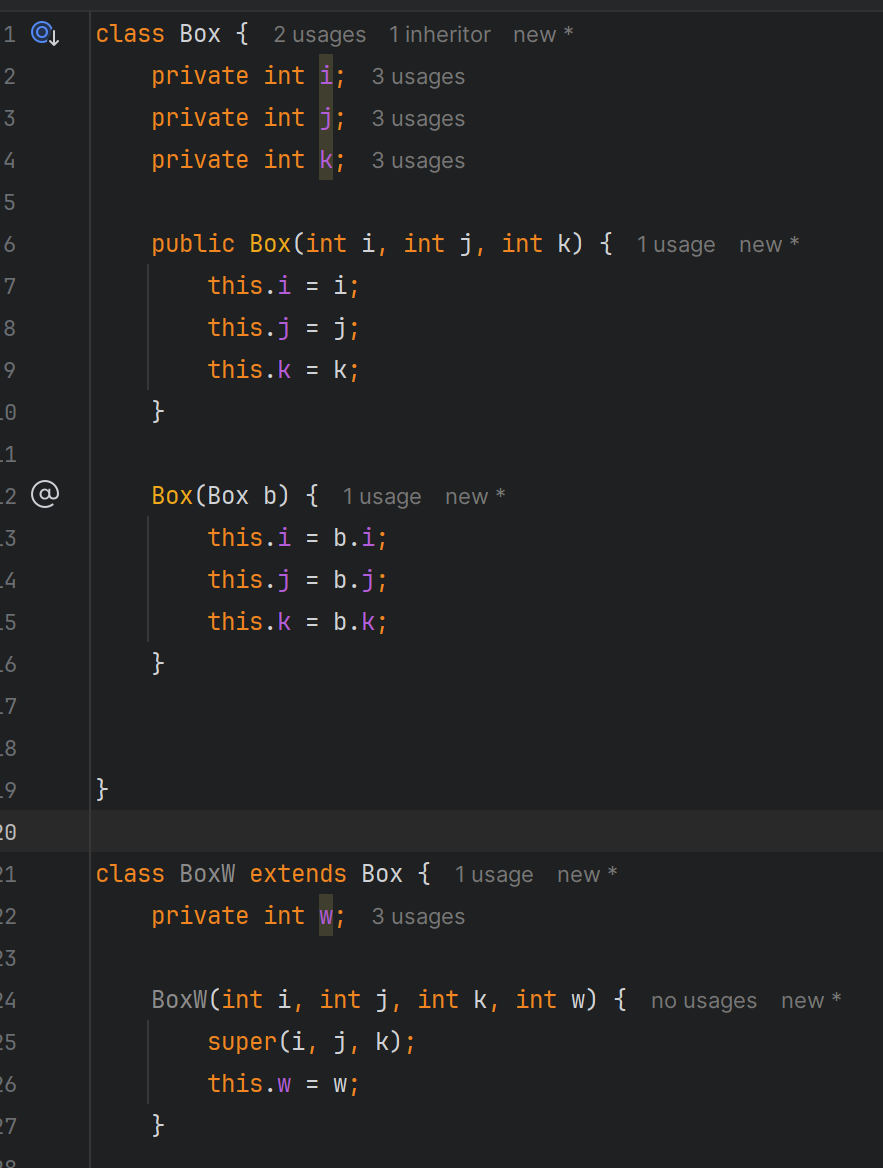
Контент, сгенерированный ИИ, может содержать ошибки.



Это для членов классов. Для самих классов только public (виден из всех пакетов) и стандартный (package-private, виден только из своего пакета). Public обязует класс иметь такое же название как и файл + быть единственным в этом файле.

Ключевое слово **super**: две формы.

1)вызов конструктора суперкласса из подкласса:



2)hiding (сокрытие) поля подкласса – имя переменной/метода в нем совпадает с тем который в суперклассе (не рек.)

Ключевое слово **final:** компилятор расознает final-блоки кода и делает «раннее связывание» (ускоряет код)

**1)именнованная константа**

**2)защита от переопределения метода**

**3)защита от наследования для класса**

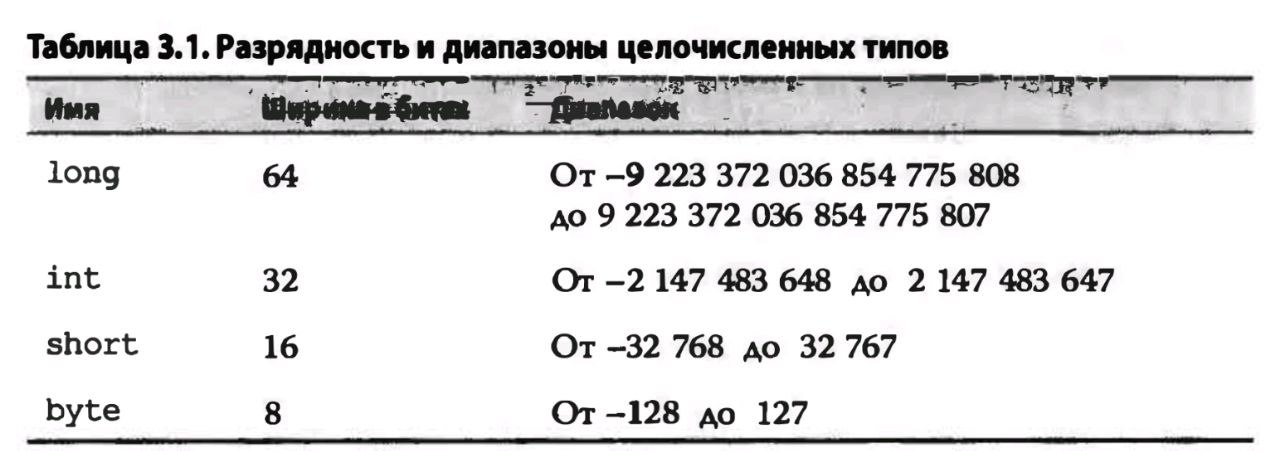
**abstract class –** только объявляет, но инициализирует методы, объект класса создать невозможно. Следует унаследоваться от такого классса и явно переопределить (@Override) все его абстрактные методы. Тем не менее, конструкторы разрешены.

**Полиморфизм** – один интерфейс для общего класса действий. «Один интерфейс, несколько методов». Один и тот же интерфейс может применяться для управления несколькими различными реализациями. Объект = экземпляр класса.

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Стандартные методы (dafault), множественная реализация (implements …), расширение интерфейса в другом интерфейсе: intarface\_1 extends interface\_2), private методы (стандартные методы интерфейса (т.е. с реализацией внутри самого интерфейса), могут использовать закрытый метод в интерфейсе (общий шаблон-метод, который можно вызывать внутри интерфейса только для методов интерфейса, но не снаружи)). Объект создавать в interface нельзя, нужно создать класс, который реализует ВСЕ методы из интерфейса (переопределять default необязательно).



**STRING**

**String** является immutable-объектом, поэтому при создании строк через литералы (“ “), разные объекты могут ссылаться на одну и ту же переменную. А значит, могут верно сравниваться через == (по ссылке). Тогда строки буду находиться в пуле строк.

При вызове методов у неизменяемой строки: string.replace(“a”, “A”), строка не изменяется. Нужно явно присвоить string = string.replace(“a”, “A”). Тогда создастся новая строка, а прошлая сотрется сборщиком мусора, так как она не будет доступна по новой ссылке.

Когда часто меняем строку, лучше использовать более производительный класс StringBuilder (работает по принципу связанного списка) только для однопоточных приложений, иначе менее произв – StringBuffer.

StringBuilder sb = new StringBuilder(100); //initial capacity (buffer)

Затем можно перевести в String: sb.toString();

Топ методы: substring (срез), toCharArray, charAt (как at в c++), indexOf (узнать индекс элемента), split (разбить строку в массив по разделителю), join (собрать строки в одну), replace (заменить все a на b), replaceFirst (только первое вхождение), toCharArray (разбить строку на массив символов, для прохода по ней)

Слово «пул» в программировании означает **набор ресурсов**, которые **перераспределяются** и **повторно используются** в процессе работы программы.

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Контент, сгенерированный ИИ, может содержать ошибки.

Для классов-оболочек есть методы парсинга. Integer.parseInt(), toString, Integer.toBinaryString (Integer.parseInt(binaryString, 2) если обратно).

**FORMATTING**

String string = “str”;

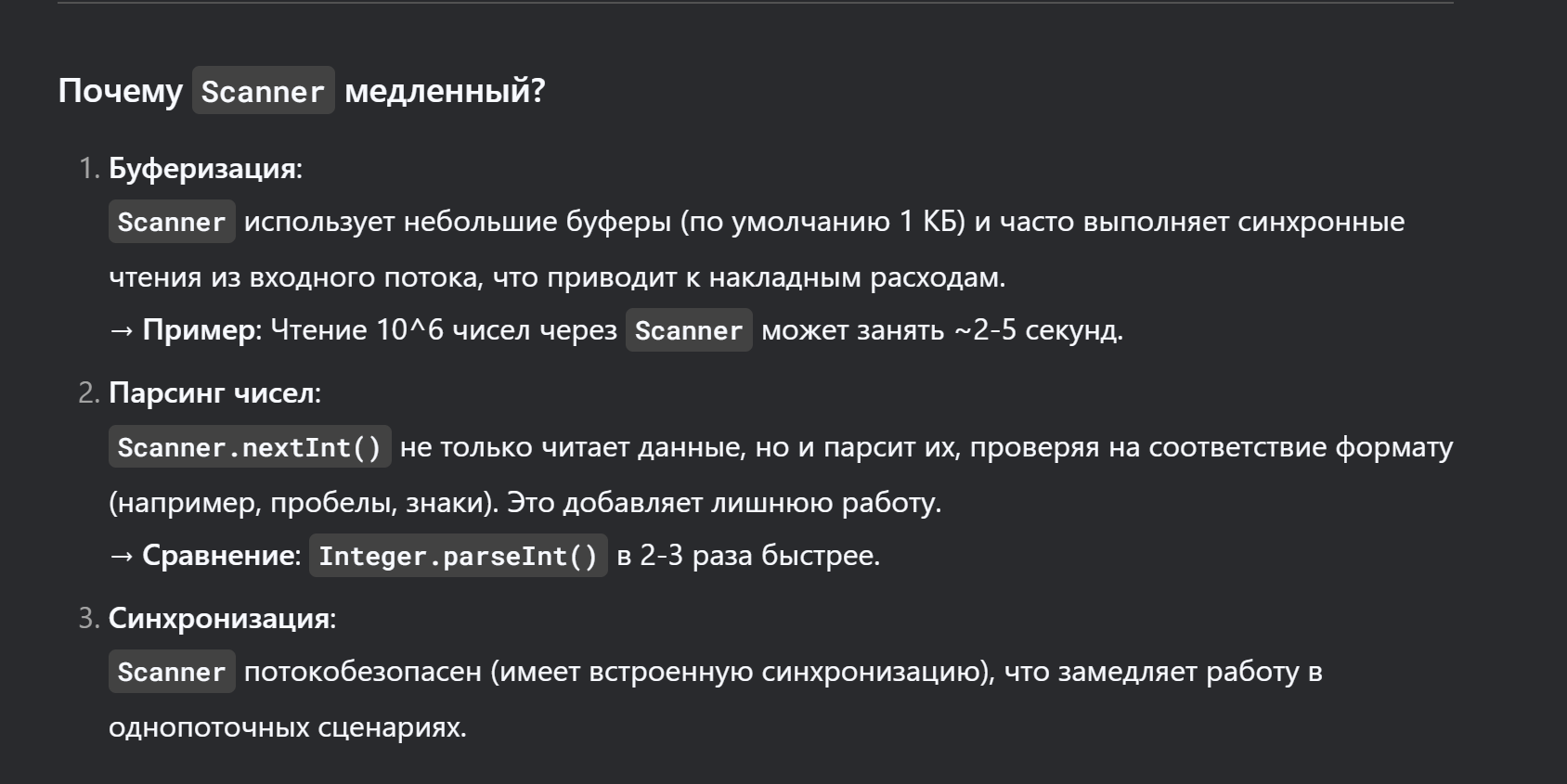
System.out.printf(“Here is a string: %s”, string);

System.out.printf(“%25s”, string) //выравнение по правому краю (вправо)

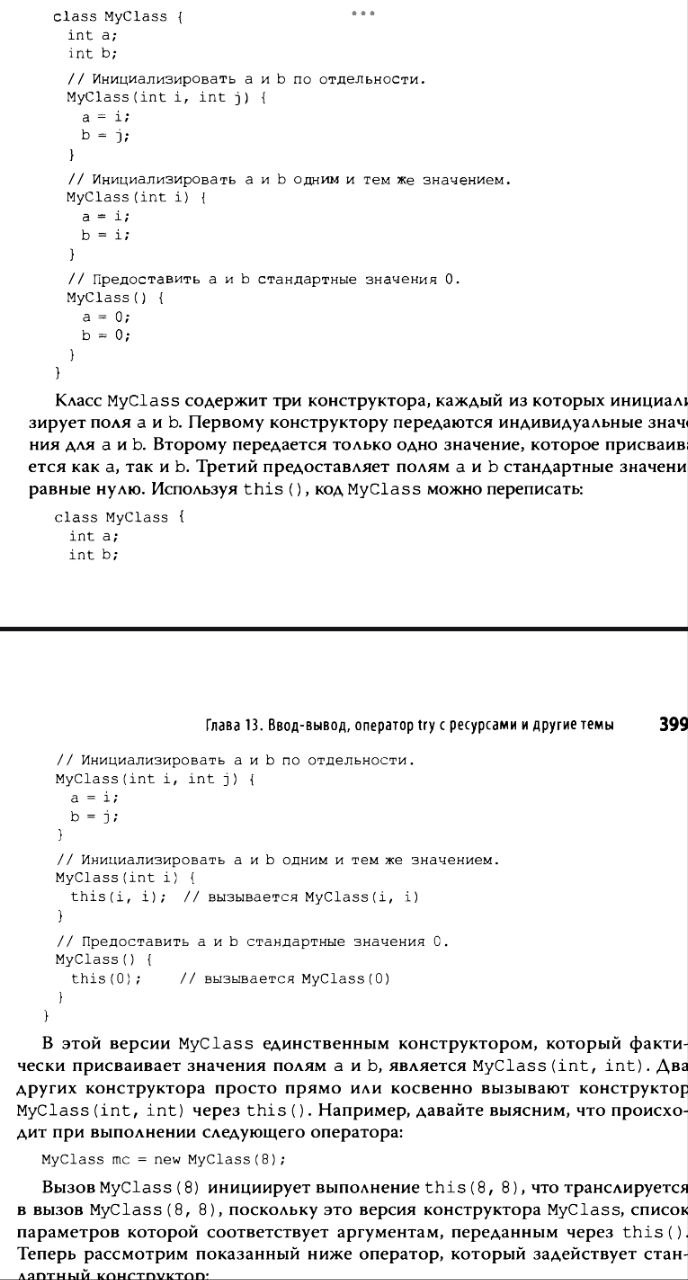
System.out.printf(“%-25s”, string) // выравнивание по левому краю (влево)

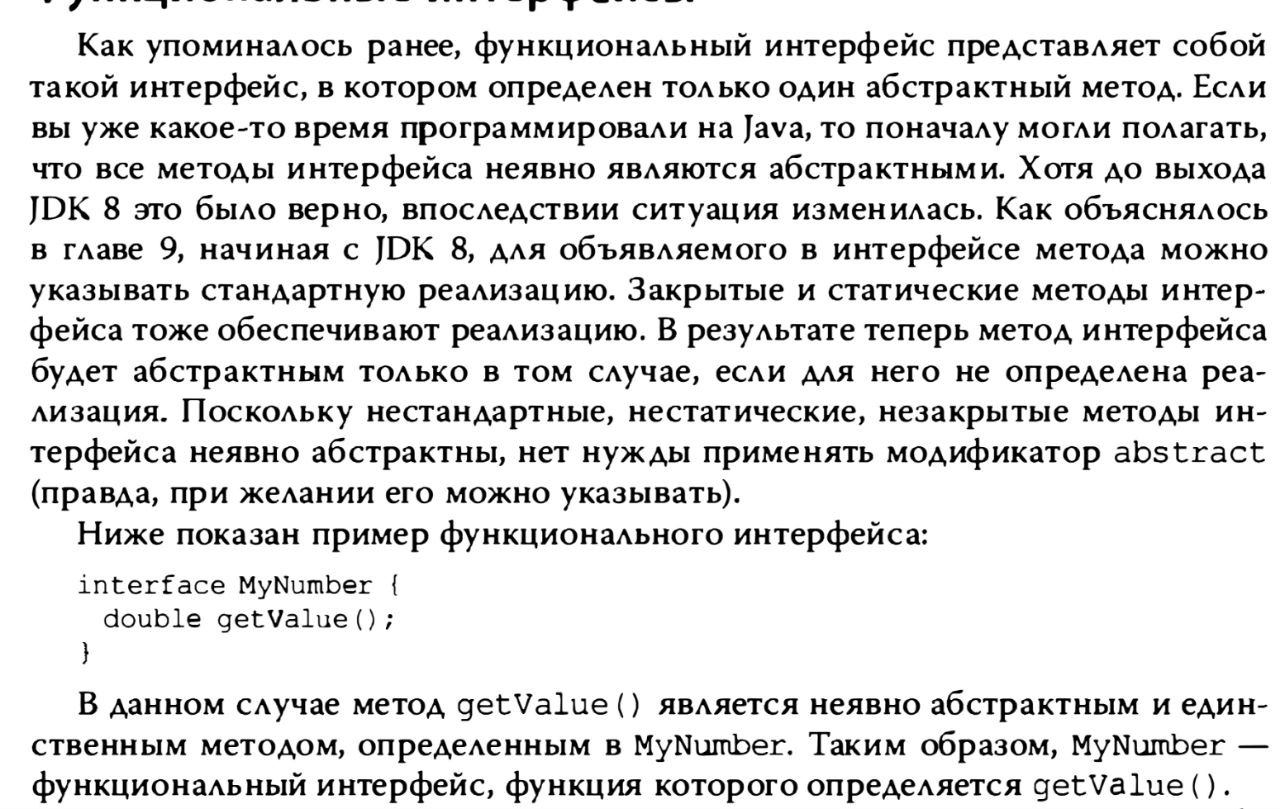
System.*out*.printf("%1.4s", str); // срез не менее 1 и не более 4 символов

System.*out*.printf("%1.2f", f); // 10,12 то есть два знака после запятой (точность)

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

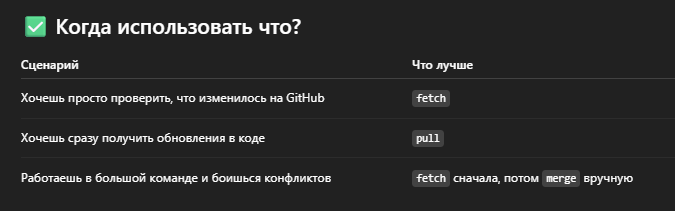
\

Эффективно когда много одинаковых полей, но для супер-большого количестав перегруженных конструкторов не рек. 

С помощью класса System можно копировать один массив в другой и смотреть время выполнения работы программы currentTimeMillis().

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

****

**CTRL+B** посмотреть исходный код (реализацию) любой бибилотеки в IDEA.

**ALT+Insert** Создать конструктор, геттеры, сеттеры

**Live Templates** psvm, fori, sout и тд.

**Local history** можно посмотреть изменения класса

**GitHub Dark theme**

**OBJECT**

Является родителем для всех классов в Джава, все наследуются от него (неявно). А значит, наследуют и все методы (половину переопределяемые, половину – final (для многопоточности)). Контракт: **obj1.equals(obj2)** **=>[obj1.hashCode() == obj2.hashCode()]**

Чтобы это работало, хеш-код должен зависеть от тех же полей.

1. **Equals()** – базовая реализация просто сранивает ссылки на объекты ( == ): this == obj. А вот если ссылки разные, это еще не означает что поля разные (бывают коллизии).
2. **getClass()** возвращает, к какому классу относится объект (в рантайме)
3. **hashCode()** возвращает число (хеш). Базово генерирует всегда разные.
4. **toString()** возвращаетинфо об объекте. Базово ИмяКласса@хеш-код.

Важно, что для одинаковых объектов всегда возвращается одинаковый хеш, а для разных желательно разный (но бывают коллизии).

Если переопределяешь equals -> обязательно переопределяешь hashCode() ! Потому что иначе HashTable сначала проверит хеши (они будут разными у разных obj1 и obj2, тк он по умолчанию генерируется независимо от полей) и сразу вернет false. В equals для final-класса getClass, иначе instanceof.

Для своих объектов **всегда** переопределяем эту пару и их контракт. Иначе p1 (20) == p2 (20) даст false.

IDEA все моментально генерирует (конструктор, геттеры+сеттеры, equals + hashcode + toString).

1)Переопределяем в своем классе equals() из Object и hashCode(). Но если вызовем obj1.equals(obj2) и obj1 = null, то будет NullPointerException.

2)Везде, где (obj1= null) возможен, используем Objects.equals(obj1, obj2) (непереопределяемый, тк static)

public static boolean equals(Object obj1, Object obj2){

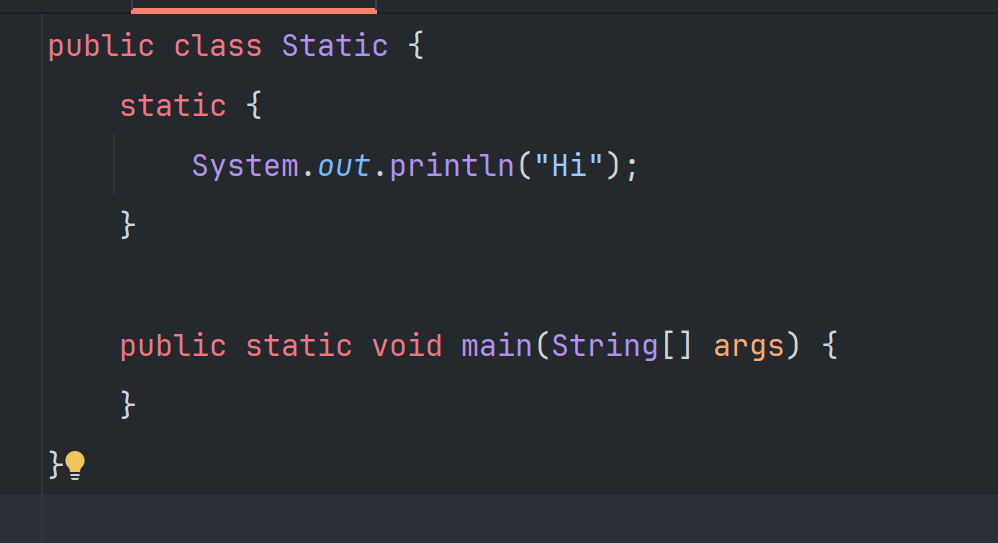
return (obj1 == obj2) || (obj1 != null && obj1.equals(obj2)).  
Здесь вначале проверяют случай (null == null) true, а затем вызывают **переопределенный** нами метод obj1.equals(obj2).

**STATIC**

Статические методы имеют утилитарный характер (Arrays.sort(), Math.max()). Вызываются у всего класса, а не конкретного объекта. Например, static в psvm – позволяет вызывать main без создания объекта класса (e.x. public class Main).

Статические переменные вызываются через класс (Object.a). Являются одинаковыми для всех его экземпляров, при создании каждого экзамплера класса это поле одинаковое для всех.

Статические блоки позволяют запускать этот фрагмент кода единожды для всех вызовов, причем сразу как создается экземпляр этого класса.



Final static int CONST = 10; Создаются один раз одновременно для всех объектов (переменная принадлежит классу) – КОНСТАНТА.

Final int a = 10; Создаются один раз но для каждого экзампляра класса различны – неизменяемые поля.

**ENUM**

Перечисления = final static переменные. (времена года, роли в игре, цвета объектов)

**ABSTRACTION**

Принцип ООП, который позволяет абстрагироваться от внутренней реализации объекта и сосредоточиться на его функциях. Например, когда пользуемся уже готовыми библиотеками, не зная их устройство внутри. «Знаем что делает объект, но не знаем как».

Абстрактный класс может содержать как обычные, так и абстрактные (только сигнатура) методы. Но если обычный класс содержит хотя бы один абстрактный метод, то он абстрактный.

Запрещено создавать экземпляры абстрактного класса, так как он абстрагирован от конкретной реализации. В абстрактном классе можно использовать обычные (не static final поля, конструкторы).

Если наследуем абстрактный класс, то обязаны имплиментировать (реализовывать) все его абстрактные методы. Можно унаследовать только один абстрактный класс, из-за отсутствия множественного наследования. Оно может привести к проблеме ромба – когда у нескольких унаследованных классов есть реализации одного и того же метода.

**INTERFACE**

Описывает методы без реализации, обычно обозначается прилагательным (Iterable, Cloneable, Comparable). Все методы всегда **публичные**. Все переменные в интерфейсах всегда константые (static final), экземпляр интерфейса создать нельзя (конструктор с параметрами тоже).

Множественное имплементирование интерфейсов позволяет избежать проблему ромба – неоднозначности, какую именно реализацию метода использовать (так как мы сами ее переопределяем в классе-потомке).

**AUTOBOXING**

Используем оболочки примитивных типов – Integer, Boolean, и тд. для преобразования примитивов в объект. Добавляет расходы за счет этих операций, не рек. злоупотреблять. Все коллекции работают с объектами: list.add(5) //автоупаковка int -> Integer

Integer a = 10; инициализируем int = 10, затем автоупаковываем в Integer.

**POLYMORPHISM**

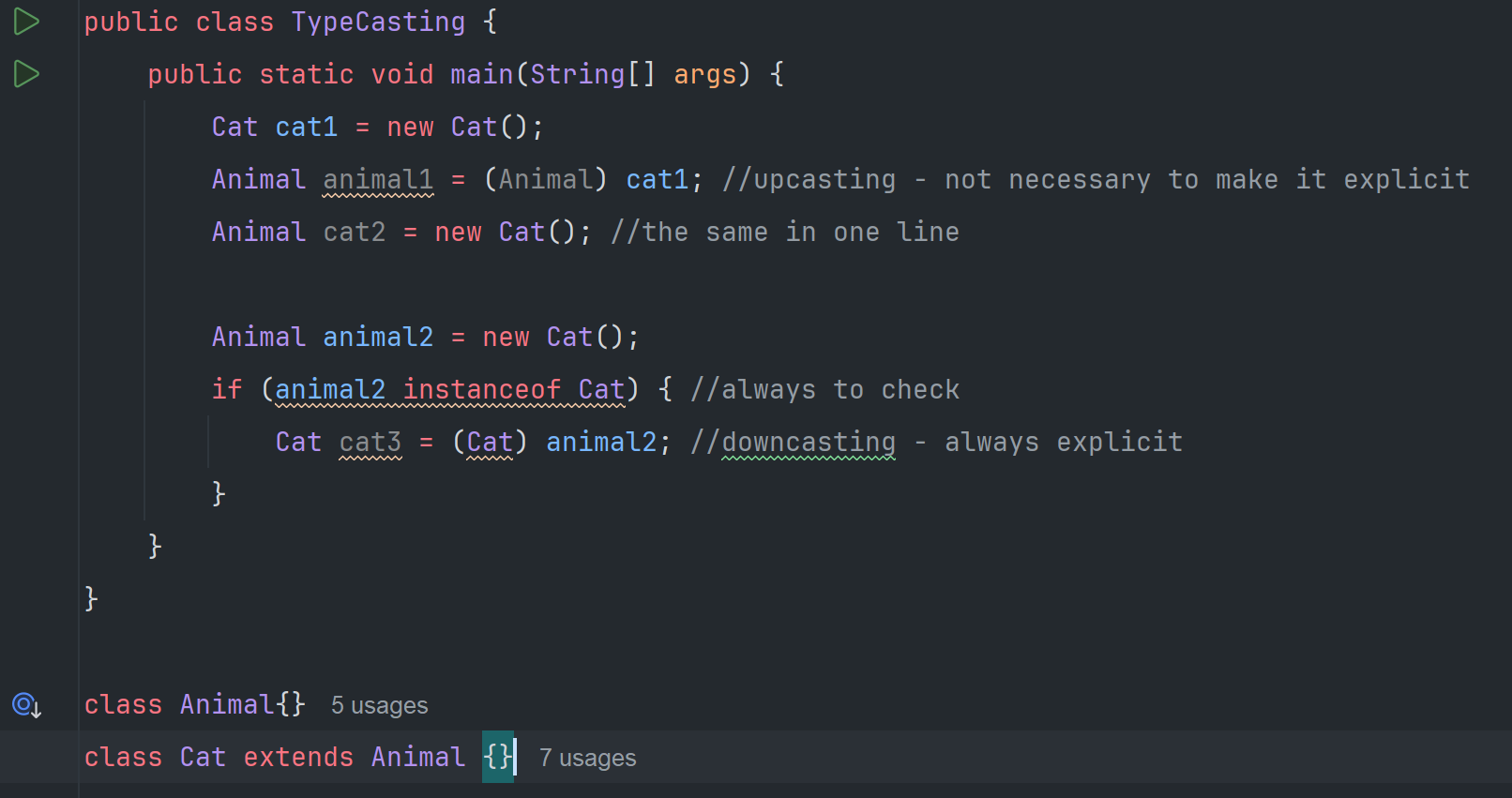
Принцип ООП, позволяющий работать с разными объектами через их общий интерфейс, а не через конкретные реалзиации. Один интерфейс – много реализаций.

Динамический полиморфизм – (run-time), с помощью переопределения.

Статический полиморфизм – (compile-time), с помощью перегрузки.

**TYPE CASTING**

Классический полиморфизм – использование общего интерфейса для разных (типов) объектов.



**GENERICS**

Параметризированный тип (класса, интерфейса, метода) – параметрический полиморфизм, позволяющий работать с классом, поведение которого не зависит от конкретного типа. Такой класс называется обобщенным (или типизированным). Обобщения работают только со ссылочными типами при объявлении, хотя передавать можно любой (будет автоупаковка).

Обобщения улучшают безопасность при несоответствии типов. Без него требовалось бы создавать класс с полями Object, а затем делать автоупаковку и явную проверку типов в runtime (через instanceof). А с обобщениями, можно во время компиляции, так как ошибки будут на этапе компиляции.

Num<Integer> number = new Num<>(10); //допустимо  
Num<int< number = new Num<>(10); //недопустимо

class Fruit<T>{}

class Bag<T extends Fruit> //ограничение на дженерик сверху

class OnlyNumbers <T extends Number> // ограничение с несколькими классами и интерфейсами можно так: <T extends Number & String>

НО class OnlyListsOfNumbers <T extends List<Number>> //compile error, иерархия нарушается (List<Integer> и List<Double> не явл наследниками List<Number>).

Для методов, в которых мы не можем сказать какой именно тип объекта мы передаем, придумали подстановочный знак (‘?’ WildCard ). public void (T<?> obj). Подстановочный знак не влияет на то, какой тип объекта T мы создаем – он лишь соответствует любому допустимоу типу объекта T. Типы объекта, в свою очередь, регулируются extends.

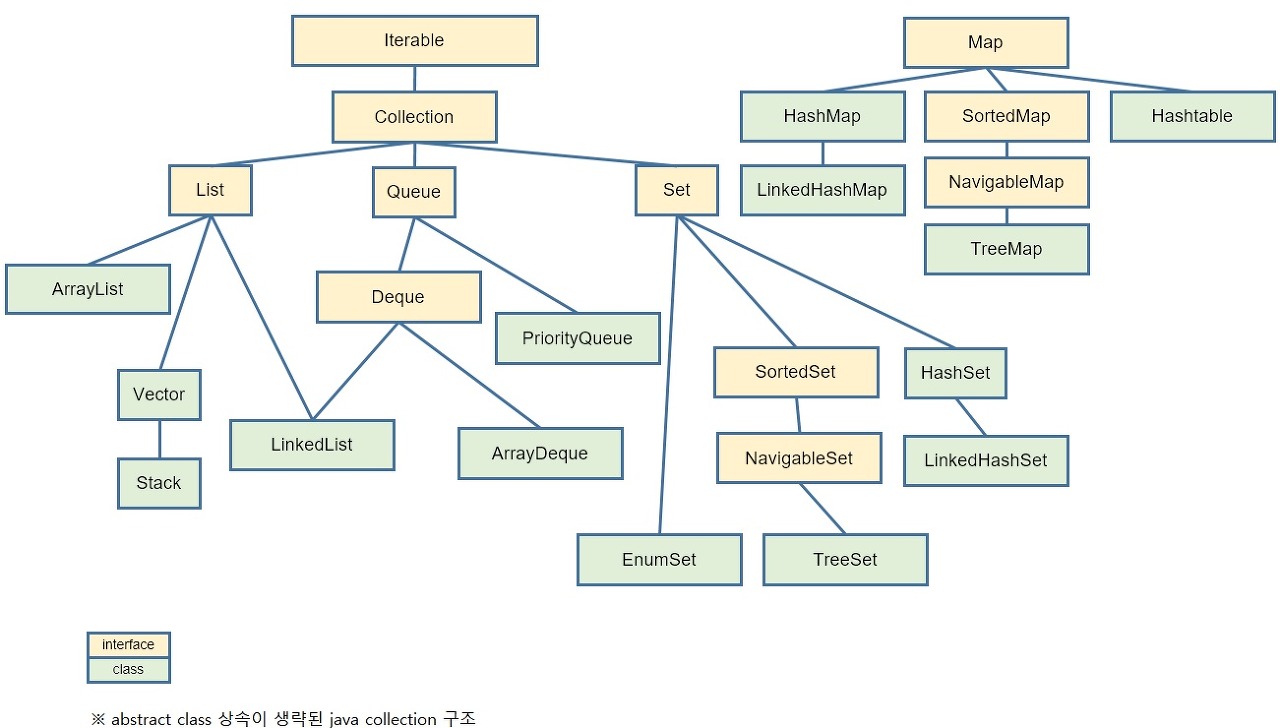
Ограниченный аргумент с подстановочным знаком: <? extends superclass>

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Дженерики нужны только на этапе компиляции, потому что тип T **стирается** до Object или верхней границы T.

**JAVA COLLECTIONS FRAMEWORK**



Изображение выглядит как текст, снимок экрана, диаграмма, линия

Содержимое, созданное искусственным интеллектом, может быть неверным.

**https://struchkov.dev/blog/ru/java-collection-framework/**

**Iterable** предоставляет метод iterator(), чтобы итерироваться по коллекции, реализующими Iterable. Метод возвращает объект Iterator, который предоставляет методы next(), hasNext(), remove(). Когда итерируемся через for-each, вызывается iterator. Но если мы **изменяем** коллекцию в одном и том же потоке (или разных), то при попытке изменить более одного элемента, вызовется ConcurrentModificationException. Это называется fail-fast iterator.

Производительная инфраструктура **Collections Framework** (по сути структуры данных). Есть интерфейсы (List, Set, Queue, Deque, Collection – гл.), и их конкретные реализации (HashSet и тп). Интерфейс Collection главный, его расширяют интерфейсы List, Set, Queue, Deque. Нигде не разрешено хранить null. Далее есть стандарные классы, которые реализуют эти интерфейсы. Также есть абстрактные классы, с помощью них можно доопределять свои коллекции.

Методы в Collection (они есть во всех реализациях): add, remove, equals, isEmpty, toArray, size.

List: список, доступ к элементу(get, indexOf), sort().

Set: не допускает дублирований. Его расширяют интерфейсы: SortedList (методы first, second), NavigableSet.

Queue (FIFO): peek() достает эл. из головы. Расширяется c пом. Deque (FIFO + LIFO): addFirst, addLast, getFirst, getLast.

Основные классы: LinkedList + ArrayList , но по умолчанию не синхронихирован для многопоточки); EnumSet; PriorityQueue; TreeSet.

**ArrayList** (дин массив, производительней чем Vector (@Deprecated), но по умолчанию не синхронихирован для многопоточки). Увеличивает/уменьшает размер по мере добавления/удаления элементов. Заранее увеличить и сократить расходы на множественное выделение памяти в последствии: ensureCapacity().

**Comparable –** когда объект имеет один и естественный способ сортировки, реализуем метод compareTo. Реализуем в своем классе этот метод. Collections.sort(people).

Реализации обоих методов должны возвращать 0, 1, -1.

public static int compare(int x, int y) {  
 return (x < y) ? -1 : ((x == y) ? 0 : 1);  
}

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

**Comparator** – (функц. Интерфейс) гибкий, когда для одного объекта можем реализовать несколько способов сортировки (например для String), реализуем несколько классов с методом compare в **отдельном классе/лямбде**. Collelctions.sort(people, (person1, person2) -> person1.name.compare(person2));

Метод compare быть реализован в функциональном интерфейсе.

Но можно не делать кастомные компараторы, так как в Comparator уже есть готовые решения:

comparingInt (для примитивов), comparing (для объектов, например String).

Можно выстраивать цепочки сравнений: сначала по возрасту, затем по имени.

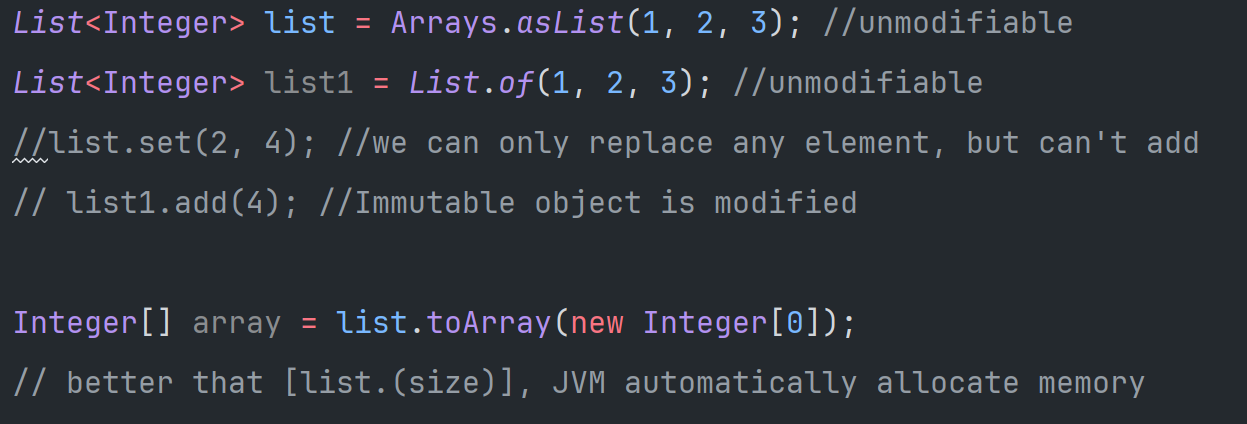
**LIST, QUEUE, SET**

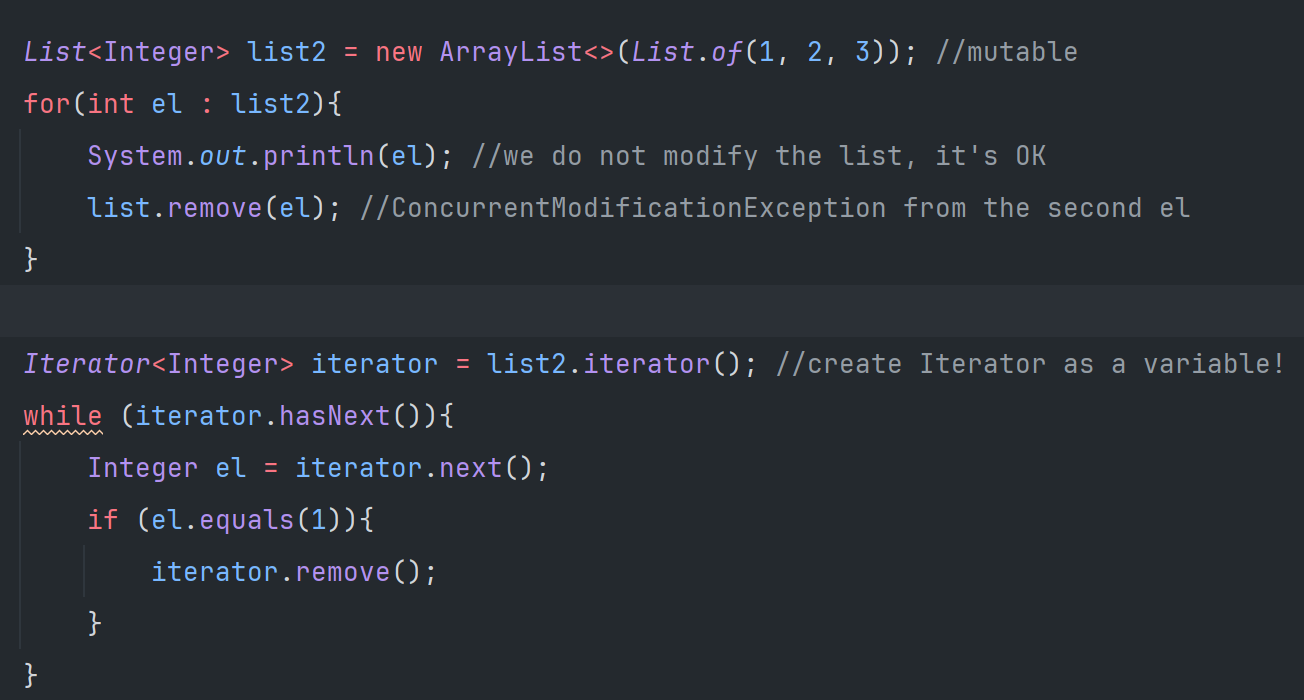
List<Integer> list = new ArrayList<>(capacity); //default capacity = 10

Dynamically extended: capacit = capacity\*1.5 + 1

Добавление в конец list.add() амортизированно O(1), тк при недостаточной емкости создается (копируется в) новый массив за O(n).

LinkedList: Джошуа Блох в “Эффективной Java” заявил, что он крайне редко оказывается производительнее ArrayList, даже для частых вставок в середину/конец/начало.





**ArrayList vs LinkedList:**

Используем ArrayList если нужно часто итерироваться по массиву, LinkedList если часто удалять из середины. Но есть сильные накладные расходы на память LinkedList.

1. Элементы динамического массива хранятся в оперативной памяти последовательно, а в двусвязном списке Nodes хаотично разбросаны
2. Для динамического массива выделяется меньше памяти (только под сами элементы), для связного хранятся Node prev, Node next, val.
3. Частый доступ по индексу за O(1) и за O(n).
4. Добавление к конец (без resize) за O(1), и за O(1).
5. Вставка в середину зачастую оптимизирована системой System.arraycopy за O(1)

Vector и Stack более не юзать (зато потокобезопасны), лучше: Deque<Integer> stack = new ArrayDeque<>();

Все коллекции не потокобезопасны, лучше использовать Concurrent Сollections (ConcurrentHashMap, CopyOnWriteArrayList, …). Не использовать synchronized-оболочки (легаси).

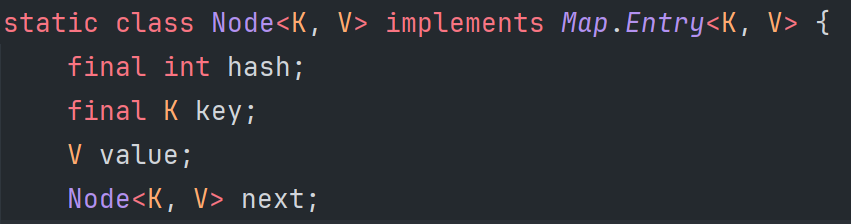
**HashSet.** Только уникальные значения.

**LinkedHashSet**. Только уникальные значения + сохраняет порядок

**TreeSet.** Только уникальные значения + отсортированный

**MAP**

**HashMap.** Представляет собой массив бакетов (ячеек), хранящий элементы Node<K, V>.





Элементы попадают в бакеты следующим образом:

1. Строится hash ключа: представляем ключ в виде числа через Object.hashCode(), затем с помощью хеш-функции преобразуем это (большое) число в меньшее, зависящее от capacity. Вычисляем индекс корзины.
2. Если корзина пустая, создаем новый Node. Если Node уже существует (Связный список не пустой)

а) Ключи совпадают – перезаписываем значение

б)Ключи расходятся – добавляем новый Node в конец связного списка

Случай, когда хеш разных ключей совпадает, называется коллизией. Плохая функция hashCode() может класть все ключи в одну корзину, тогда операции вырождаются в O(n) по связному спискую. В таком случае Node добавляется в конец связного списка в этом бакете. Элементы в таком случае не теряются, но находить их становится дольше – за O(n).

Чтобы распределение хешей было равномерным, таблица переодически расширяется в 2 раза за счет рехеширования всех ключей. За это отвечает loadFactor.

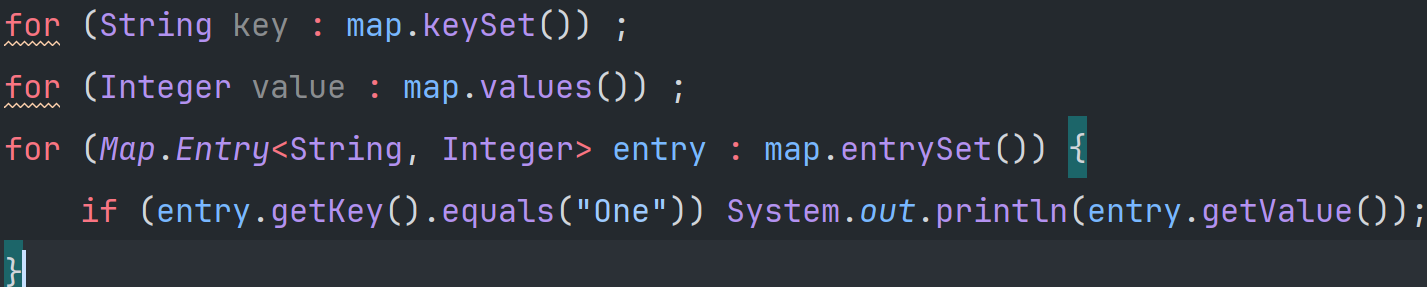
public HashMap(int capacity, float loadFactor). Default capacity=16, loadFactor = 0.75

Capacity всегда = 2^n. Когда в одном бакете содержится >= 8 Node, список преобразуется в красно-черное дерево, где операции поиска за O(logn). Обратно, если становится <= 6, преобразуется в список (за O(n) туда и обратно).

Для ключей, являющихся объектами собственных классов, обязательно переопределять equals и hashCode (оба).

Нельзя использовать в качестве ключей mutable-объекты, хеш код ключа изменится и элемент станет недоступен, тк будет находиться по старому адресу.

**HashMap.** Не сохраняет порядка элементов. Разрешает один null ключ (всегда хранится в нулевом бакете), множество null значений. Не потокобезопасна + работает по типу fail-fast, т.е. выдает исключение при любой несогласованности данных -> лучше ConcurrentHashMap. HashTable синхронизирована, но не принимает null ключи. Реализован с помощью метода цепочек (связанный список в бакете). Еще есть открытая адресация (линейное, квадратичное, двойное пробирование).



**LinkedHashMap.** Сохраняет порядок добавления элементов

**TreeMap.** Отсортированный, по умолчанию по ключу (стандартно, если String, Integer)

Если будет добавляться null, то нужно явно указать куда он попадет (в начало или в конец), затем в его скобках основной компаратор:

*Map*<String, Integer> map = new TreeMap<>(*Comparator*.*nullsFirst*(*Comparator*.*naturalOrder*()));

**EXCEPTIONS**

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Содержимое, созданное искусственным интеллектом, может быть неверным.

Проверяемые исключения – вылетают сразу же, на этапе компиляции (критические ошибки), нужно обязательно отслеживать.

Непроверяемые исключения – программа компилируется, но вылетает исключение во время исполнения, не нужно отслеживать.

Error: IOError

RuntimeException: IllegalArgumentException, NullPointerException (невозможно отследить все), ArithmeticException, IndexOutOfBoundsException, ConcurrentModificationException.

Other exceptions: IOException, FileNotFoundException, SQLException (файлы, сеть, БД).

Не нужно все исключения отлавливать типом Throwable e – мы не будем знать какое исключение выпало.

**Stack Trace** - Ошибка, которая пишется в консоли, если нет блока try-catch. Иначе указывать вручную – e.printStackTrace. Структура вывода (первым – место, где программа завершилась): Поток – Ошибка – Конкретика ошибки

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

e.getStackTrace – предоставляет массив из имени класса, метода, номера строки.

Note: 10\0 – ArithmeticException, 10.0\0 – Infinity (вывод).

Для создания своих исключений – наследуемся от Exception (будет проверяемым) или от RuntimeException (непроверяемым). Нужно будет переопределить конуструкторы родителя (необходимое колво от 1 до 4).

**Throw new** создает объект исключения и выбрасывает его (например, в if). Поэтому некоторые исключения нужно импортировать (их классы).

Многие исключения из java.lang (не нужно импортировать, авто).

Из проверяемых: java.io, java.net, java.sql;

Если выбрасываем через throw проверяемое исключение, то метод, который его выбрасывает, ОБЯЗАТЕЛЬНО должен указать это в своей сигнатуре через throws.

**Finally** – блок выполнится в любом случае: если исключение не было выброшено, либо было выброшено. Но system.exit() остановит работу JVM – единственный случай невыполнения finally.

Используем это ключевое слово для закрытия потоков вывода/ввода: например, Scanner с потоком ввода не System.in().

Лучше не добавлять return в finally (иначе всегда возвращается одно значение).

Если не писать catch, то исключение будет отловлено сразу после выполнения finally, в этом же блоке.

**РАБОТА С ФАЙЛАМИ**

Сначала (до 7 версии) использовали java.io.File, теперь более новый API – java.nio.file.Path, потому что в первоначальном варианте часто возвращались false вместо исключения + нет метода copy.

Вместо единого класса File, создали два класса (Files + Paths) и интерфейс Path.

Класс Paths предоставляет только два метода only for cast: URI -> Path, String -> Path.

Интерфейс Path предоставляет методы для работы с путями, поиска корней и тд.

Класс Files предоставляет статические методы для работы с файлами из File (+copy()).

**STREAMS**

Поток – бесконечная последовательность данных (символьный или байтовый). Все классы для работы ввода/вывода находятся в java.io

Изображение выглядит как текст, снимок экрана, диаграмма, линия

Содержимое, созданное искусственным интеллектом, может быть неверным.

Самым распространенным источником/получателем информации является File: FileInputStream и FileOutputStream (pdf, png, хотя можно и текстовые), FileReader и FileWriter (текстовые).

Эти четыре абстрактных класса могут выбросить исключение IOException (проверяемое!), поэтому нужно объявлять это исключение в сигнатуре метода через throws.

**Reader** считывает текст **посимвольно.** Метод reader.read() возвращает значение int, которое содержит значение прочитанного символа char. Конец файла можно проверить так:

int symbol = reader.read();

while (symbol != -1)

**BUFFERED**

**Буферизация –** способ ввода/вывода, при котором данные временно сохраняются в промежуточной области памяти (буфере).

BufferedReader и BufferedWriter (текст), BufferedInputStream и BufferedOutputStream (фото + текст (не рек.)).

У Reader есть метод read() – считывает посимвольно как int, и readline() – считывает строку целиком как String (проверяем на null).

BufferedReader bf = new BufferedReader(new FileReader(“text.txt”));

BufferedWriter bf = new BufferedWriter(new FileWriter(“text.txt”));

В конструкторах (их 2), нужно передавать поток ввода/вывода. Во втором конструкторе можно передать вторым параметром размер буфера (по умолчанию 8192).

С любым классом из java.io нужно объявлять в сигнатуре метода throws IOException (чтение/запись в файл).

Потоки нужно всегда закрывать после использования **bf.close()**, чтобы освободить ресурсы операционной системы, потому что любой поток (=>ресурс) реализовывает интерфейс Closeable или AutoCloseable.

**НО:** scanner.close() делать не нужно (если поток System.in()), иначе консольный ввод закроется на протяжении всей программы.

До 7 Java нужно было вручную проверят потоки на null и обрабатывать все в try-catch-finally.

Изображение выглядит как текст, снимок экрана, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Затем появилась конструкция **try-with-resources**, которая автоматически закрывает ресурсы, реализующие Closeable и AutoCloseable, сразу после заврешения это блока (независимо от всего).

Изображение выглядит как текст, снимок экрана, программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

Объекты классов Writer автоматически создают файл при объявлении и открывают его для записи/чтения. Если мы не создавали объекты Writer, а создали только Reader и файла нет – выбросится FileNotFoundException.

Чтобы не пересоздавать файл каждый раз, можно писать в уже созданный файл. Для этого достаточно добавить второй параметр **true** в FileWriter:

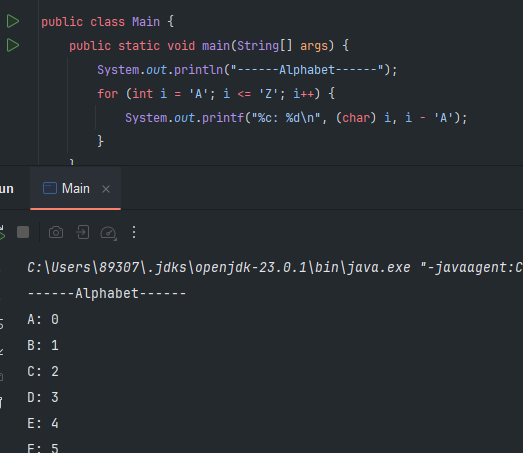
BufferedWriter bf = new BufferedWriter(new FileWriter(path, **true**))

FileNotFounsException является подклассом IOException, поэтому автоматически перехватывается при catch (IOException e), но иногда полезно перехватывать исключение “File not found”, тогда нужно отлавливать сначала FileNotFoundException, а затем IOException (по возрастанию иерархии), чтобы отловить остальные.

**ENCODING**

Unicode (0-127, 0-32 управляющие).

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

**ANONYM CLASSES**

-Объявляет и сразу инициализирует объект, реализующий интерфейс (или класс).

-Не может иметь конструктора, так как не имеет явного имени

-Не может быть static, так как привязан к конкретному объекту

-Переопределяет методы (можно и поля) родительского класса или интерфейса, но не может добавлять новые

-Можно сохранить в переменную: Runnable run = new Runnable() и вызывать методы run. , но только в пределах метода, в котором объявлен анонимный класс.

Резюме: вместо того чтобы создавать отдельный класс (реализующий интерфейс/переопределяющий методы класса), уже в самом методе создаем “мини-класс”, который тут же используем на месте + интерфейс/родительский класс.

В библиотеке java.util.function находятся уже готовые функциональные интерфейсы: Supplier, Consumer, Function, Predicate.

Изображение выглядит как текст, снимок экрана, программное обеспечение

Содержимое, созданное искусственным интеллектом, может быть неверным.

**LAMBDA**

1. **Структура.**

Входные параметры -> реализация функционального интерфейса.

(String text) -> { return Character.isUpperCase(text.charAt(0)); }

1. **Правила синтаксиса.**

Лямбда может иметь 0 и более входных параметров.

Параметры заключаются в круглые скобки и разделяются запятыми (String country, int size) -> { … }

Если тип параметра можно получить из контекста, его можно не указывать (country, size) -> { … }

Если параметр один и его тип явный, скобки можно опустить.

distance -> return distance / 100;

1. **Отсутствие параметров:**

Если параметров нет, ставятся пустые скобки.

() -> "Магазин закрыт";

1. **Тело лямбды:**

Тело лямбды может состоять из 1 или более выражений.

Если тело состоит из одного оператора, фигурные скобки и слово return можно опустить.

distance -> distance / 100;

1. **Возврат значения:**

Если тело состоит из нескольких операторов, нужно использовать фигурные скобки и явный возврат через return.

(country, size) -> {

return country + size;

}

Лямбда может реализовывать только функциональный интерфейс (с 1 абстрактным методом).

Также лямбда не может содержать своих (новых ) полей.

Лямбда может обращаться к локальным переменным (или параметрам) для их чтения, если такие переменные финальные или эффективно финальные (явно не являются final, но не изменяются после инициализации переменной).

Int x = 10;

x = 20; //x is not more an effective final variable!

Runnable r = () -> System.out.println(x); // compilation error

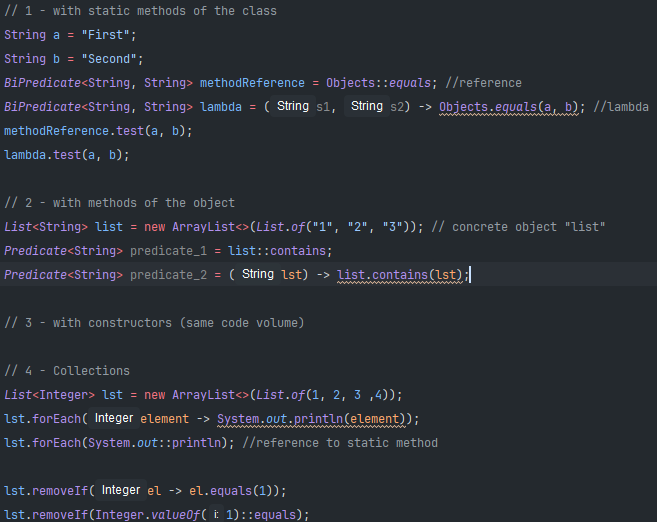
r.run();

Так происходит, потому что локальные переменные (в методе) хранятся в стеке (временная память) и стираются после работы метода. Лямбда генерируется в рантайме и запоминает только ссылки этих переменных, поэтому не может их изменить/обратиться к изменившимся переменным.

Такое свойство называется **замыканием**, когда лямбда запоминает ссылку на переменную, даже когда та очищается GC.

Лямбда может обращаться к любым полям класса и изменять их.

Для коллекций удобно использовать метод .forEach.



**OPTIONAL**

Если может появиться значение null -> может вылететь NullPointerException -> нужно использовать значение не напрямую, а через обёртку в виде объекта Optional<>.

Обертка Optional помогает избежать ошибок с null и предоставляет методы для безопасной работы с ними. То есть, этот класс следует использовать, когда мы предполагаем получить null.

obj.ifPresent(), Optinal.of(obj), Optional.empty() – пустая обертка

System.*out*.println(Optional.*ofNullable*(c1).equals(Optional.*ofNullable*(c2)));

Но на практике это избыточно для проверки равенства, можно просто == null, но лучшее решение – Objects.equals(), который сначала проверит

(null == null) случай, а затем вызовет наш переопределенный obj1.equals(obj2) (заранее проверив obj1!=null), чтобы смогли вызвать этот метод.

**STREAM API**

Напоминает коллекции, но работает с уже существующими данными, единоразово.

*Stream*<String> stream\_1 = *Stream*.*of*("first way"); //--1--  
*List*<String> list = new ArrayList<>(*List*.*of*("second way")); //--2--  
*Stream*<String> stream\_2 = list.stream();

Операции со стримом бывают двух видов – **промежуточные** (intermediate), которые выполняются последовательно, и **терминальные**, которые выполняются только один раз. Создавать отдельную переменную для стрима не обязательно, промежуточные операции вызываются через точку.

list.stream().intermediateOperation1().intermediateOperation2;

list.stream().forEach(…) // то же самое, что list.forEach(System.out::println)

Использовать можно только один раз, в одной строке.

**Терминальные операции**: count (кол-во элементов), allMatch (проверка условия для каждого элемента), anyMatch (есть хотя бы один нужный элемент), forEach (для прохода по каждому), collect (собирает в коллекцию).

**Промежуточные операции:** filter (Predicate, принимает предикат и возвращает подходящие элементы), map (Function, принимает функцию и возвращает новые элементы).

Несколько условий лучше разбивать на последовательный вызов двух функций, чем на &&-условия.

**TIME**

Unix-Epoch – точка начала отсчета (1970 год).

Instant – храним инфо о моменте из прошлого или будущего.

LocalDateTime – храним текущую дату и время с помощью метода now(), причем берем это именно с самого устройства. Можно управлять отдельно с помощью классов LocalDate и LocalTime.

Period и Duration – длительность промежутков в днях и наносекундах соответственно (разная точность).

ZonedDateTime -привязка к конкретному числовому поясу (+смещения), региону

**NETWORK PROTOCOLS**

Client-Server. Request-Answer. Protocol, Stack of Protocols (Hierarchy)

Internet Protocol (**IP**): присваивается каждому устройству индивидуально, не ПК, а приложения. Поэтому для доставки инфо нужно знать номер приложения – **порт**. Номера портов на клиенте не фексированные (номера > 1024, в отличие от сервера (обычно номер до 1024), а назначаются ОС динамически. В сетях работающий по протоколам TCP/IP файлы передаются пакетами, которые содержат сами данные и адреса получателя/отправителя + служебное инфо.

DNS, порты, через cmd (netstat -an, udp, ftp, почтовый, прокси, маска, шлюзы, уровни

Набираем URL (доменное имя), получаем страницу сайта.

Transfer Control Protocol (**TCP**): Транспортный уровень (добавляет проверку доставки), проверяет целостность IP-ответа и конкретную программу с помощью портов.

UDP

FTP

HyperText Transfer Protocol (**HTTP**): использует IP и TCP для взаимодействия между HTTP-Client и HTTP-Server.

**URL**

<http://vk.com/hehe_b0y>

Протокол://сервер/местоположение

Для обращения к серверу через HTTP, который использует IP/TCP протоколы, нам необходимо знать IP-адрес сервера и его сетевой порт. Но писать наборы символов неудобно, поэтому каждому IP представлен его DNS (система доменных имен). Также, HTTP серверы обычно используют порт 80, поэтому его часто не пишут.

**HHTP-request** **HTTP-answer**

HTTP/1.1 200 OK

Accept-Ranges: bytes

Connection: close

Content-Length: 2217

Content-Type: text/html

Date: Wed, 24 Feb 2021 08:02:57 GMT

ETag: “e23432500”

Last-Modified: Thu, 03 Dec 1992 08:37:28 GMT

Server: Apache

<HEADER>  
<TITLE> The World Wide Project </TITLE>

…

(another info on page)

GET /hyperText/WWW/TheProject.html HTTP/1.1

Accept: \*/\*

Accept-Encoding: gzip, deflete

Connection: keep-alive

Host: info.cern.ch

User-Agent: HTTPie/2.3.0

Сначала идет стартовая строка, затем Заголовки, затем Тело.

Стартовая строка: описываем метод для сервера, сервер отправляет код ответа (status code), а также оба возвращают отправляют версию протокола HTTP.

Заголовки (доп инфа) – accept (тип данных, которые принимаем), Host (ресурс), Content-length (размер тела сообщения в байтах), Content-type (тип данных, которые передаем).

**HTTP-REQUEST METHODS**

**POST** - создание на сервере нового ресурса, e.x. добавление нового пользователя в БД, но если такой ресурс уже есть – сервер вернет ошибку.

**GET** - получение ресурса, e.x. о созданном польвателе, в ответ на такой запрос сервер вернет польз. данные.

**PUT** - используется, как и POST, для передачи данных серверу. Отличие в том, что PUT-запрос заменяет существующие данные или при их отсутствии создает новый ресурс, без ошибки.

**PATCH** – частично обновляет данные ресурса (только некоторые данные пользователя).

**DELETE** – удаление ресурса.

**STATUS CODES**

Трехзначные коды, которые возвращает в статусной строке сервер.

**Коды успеха (200-299):** 200 (OK) – запрос успешно обработан, и результат возвращен в теле ответа, 201 (Created) – на сервере успешно созданы ресурсы, 204 (No content) – запрос успешно обработан, но для возврата нет никаких данных, тело ответа обязательно будет пустым.

**Коды клиентских ошибок (400-499):** запрос сформулирован некорректно (e.x. содержит неверные данные или не может быть выполнен). 400 (Bad Request) – сервер не понимает запрос или пытается его обработать, но что-то неверно, 401 (Unauthorized) – для выполнения запроса нужна аутентификация, но вместе с запросом не было отправлено автозризационных данных. 404 (Not Found) ресурс не найден.

**Коды серверных ошибок (500-599):** сервер не смог обработать запрос из-за внутренних проблем. 500 (Internal Server Error) – у сервера возникла

Проблема, с которой он не справился, поэтому сбой обработки, 501 (Not Implemented) – серверу неизвестен HTTP-мтеод, использованный в запросе, поэтому запрос невозможно обработать, 503 (Service Unavailable) - сервев не может сейчас обработать запрос из-за сильной загрузки/перезагрузки.

**HTTP-СЕРВЕР**

**Web Service** – серверы, обрабатывающие запросы фронтенда.

WEB **API** – набор URL-адресов и соглашений об их использовании.

**JSON**: frontend – передача данных на сервер в теле POST, PUT, PATCH запросов, backend – тело ответов (GET и PUT).

**Endpoint** – сочетание URL (неполного) и HTTP-метода. За его описание отвечат HTTP-сервер. Пример эндпоинта: GET /photos/my\_photo

Вся информация передается блоками, которые называются **network packet** (сетевой пакет). Внутри пакета содержатся не только данные, но и IP-адрес (адрес) и номер порта (программа-получатель).

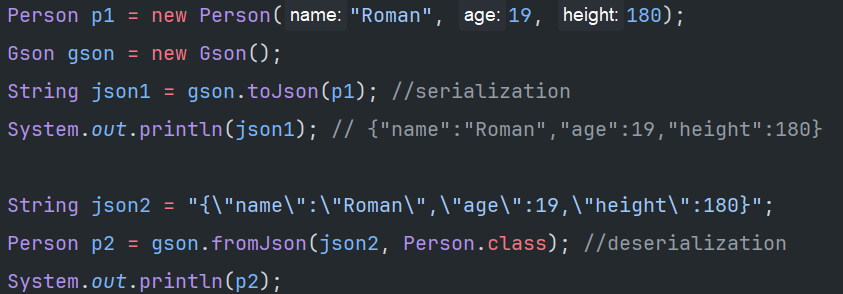
**Сокет** – средство ОС, с помощью которого программа может получать доступ к сети. Процесс получения доступа называется **открытием сокета.** IP-адрес и номер сетевого порта – параметры сокета. Одновременно могут быть открыты несколько разных сокетов, так как у них могут быть разные IP и порты. Когда сокет открыт, программа «слушает порт», то есть может обмениваться запросами.

**HTTP-сервер:** существует два способа создания. В обоих методах мы указываем номер порта backlog (число входящих соединений, которые могут ожидать своей очереди, если вписать 0 – будет кол-во по умолчанию ОС). IP-адрес не указываем – сервер будет искользовать любую доступную на компьютере сеть.

Изображение выглядит как текст, снимок экрана

Содержимое, созданное искусственным интеллектом, может быть неверным.

Сериализация – процесс трансформации Java-объекта в какой-то другой формат (обратно – десериализация). Используем модуль Gson.



По правилам JSON, все ключи и строковые значения должны быть заключены в двойные кавычки. По умолчанию – без пробелов, отступов и переносов строк. Это делается для ускоренной передачи объектов в единицу времени, существенно экономит ресурсы.

Для удобства, есть множество настроек для Json с помощью GsonBuilder:

GsonBuilder gsonBuilder = bew GsonBuilder();

gsonBuilder.setPrettyPrinting(); // for human-readeable printing in console

gsonBuilder.serializeNulls(); // **default JSON does not contain (=>print) null**

Gson gson = gsonBuilder.create();

String json = gson.toJson(obj);

Чтобы возвращать/принимать данные в другом (своем или особенном) формате, нужно реализовать сериализатор (десериализатор) со своими заданными параметрами и вызвать метод registerTypeAdapter. Написать свой конвертер можно унаследовавшись от TypeAdapter. Это нужно, потому что такие объекты не могут быть сериализованы в json стандартным способом.

Для разработки и тестирования HTTP запросов используем Postman.