Java с нуля

Java с нуля

Содержание

- Метод clone
- Принципы ООП
- Наследование и Ассоциация (определение, плюсы-минусы каждого)
- Переопределение методов
- Статическое и динамическое связывание
- Понятия
  Рефлексия в Java

Модуль 1

Какие языки программирования бывают

Языки программирования — это формальные языки, предназначенные для написания инструкций, которые компьютер может понять и выполнить. Они классифицируются по различным критериям: по уровню абстракции (низкоуровневые и высокоуровневые), по парадигмам (процедурные, объектно-ориентированные, функциональные и т.д.), по назначению (веб-разработка, мобильные приложения, анализ данных и т.д.). Вот основные категории с примерами:

- **Низкоуровневые языки**: Близки к машинному коду. Примеры: Ассемблер (для прямого управления аппаратными ресурсами, как в embedded-системах). Плюсы: высокая производительность, контроль над памятью. Минусы: сложность, нечитабельность.
- **Высокоуровневые языки**: Абстрагируют детали аппаратного обеспечения. Примеры: Python (для скриптинга, AI, веб), Java (для enterprise-приложений, Android), C++ (для игр, системного ПО), JavaScript (для веб-фронтенда и бэкенда).
- **По парадигмам**:
  - Процедурные: Фокус на функциях и процедурах. Примеры: C, Pascal.
  - Объектно-ориентированные (ООП): Организация кода вокруг объектов и классов. Примеры: Java, C#, Python. Здесь используются принципы инкапсуляции, наследования, полиморфизма и абстракции.
  - Функциональные: Трактуют вычисления как математические функции, избегая изменяемого состояния. Примеры: Haskell, Scala, Lisp.
  - Мультипарадигменные: Поддерживают несколько стилей. Примеры: Python, JavaScript.
- **По назначению**:
  - Веб-разработка: HTML/CSS (не языки программирования, но markup), JavaScript, PHP, Ruby.
  - Мобильная разработка: Swift (iOS), Kotlin/Java (Android).
  - Data Science: R, Python (с библиотеками как Pandas).
  - Системное программирование: C, Rust (фокус на безопасности памяти).

Языки эволюционируют: например, Java сочетает ООП с функциональными элементами с версии 8. Для изучения рекомендую официальные источники, такие как документация Oracle для Java или Python.org для Python. Подробнее о классификации: [web_search для "programming languages classification site:geeksforgeeks.org" даст хорошие обзоры].

JVM, JRE, JDK

Эти термины связаны с экосистемой Java и обеспечивают выполнение Java-кода на разных платформах (принцип "write once, run anywhere" — часть абстракции в ООП).

- **JVM (Java Virtual Machine)**: Виртуальная машина, которая выполняет байт-код Java (скомпилированный .class-файлы). JVM интерпретирует байт-код в машинный код для конкретной ОС. Она управляет памятью, garbage collection, безопасностью. Без JVM Java-программы не запустятся. Пример: Когда вы запускаете `java MyClass`, JVM загружает классы и исполняет main-метод.
- **JRE (Java Runtime Environment)**: Среда выполнения, включающая JVM и стандартные библиотеки (как java.lang). JRE нужна для запуска Java-приложений, но не для разработки. Если у вас есть .jar-файл, JRE его запустит.
- **JDK (Java Development Kit)**: Полный набор для разработки, включающий JRE, компилятор (javac), отладчик, инструменты (javadoc) и API. JDK нужен для написания и компиляции кода. Пример: `javac MyClass.java` компилирует в байт-код с помощью JDK.

Установка: Скачайте с официального сайта Oracle или Adoptium. Пример команды для проверки: `java -version` (покажет JRE/JVM), `javac -version` (JDK). Ссылки: Официальная документация Oracle — https://docs.oracle.com/en/java/javase/21/intro/what-is-java-technology.html (для Java 21, актуально на 2025).

Области памяти в Java

Память в Java делится на области, управляемые JVM. Это обеспечивает безопасность и автоматическое управление (garbage collection — часть абстракции).

- **Heap (Куча)**: Для динамически выделяемых объектов (new). Здесь хранятся экземпляры классов. Делится на Young Generation (Eden, Survivor) для новых объектов и Old Generation для долгоживущих. Garbage Collector очищает неиспользуемые объекты.
- **Stack (Стек)**: Для локальных переменных, ссылок на объекты, вызовов методов. Каждый поток имеет свой стек. При переполнении — StackOverflowError.
- **Metaspace (ранее PermGen)**: Для метаданных классов, статических переменных, констант. Неограничен по умолчанию, но может вызвать OutOfMemoryError.
- **Native Memory**: Для JNI (Java Native Interface), не управляется GC.

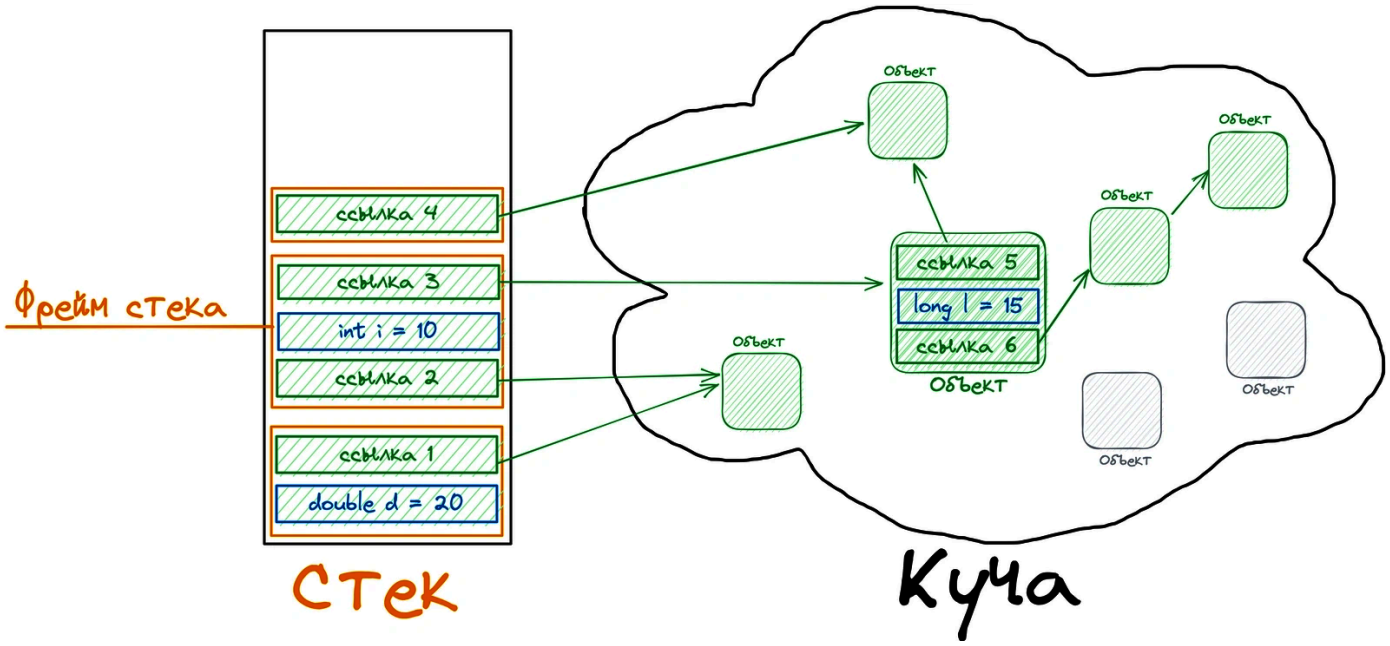
Пример: Локальная переменная `int x = 5;` в стеке, объект `String s = new String("hello");` — ссылка в стеке, данные в heap. Подробнее: Baeldung — https://www.baeldung.com/java-memory-

management.

Куча (Heap)

Куча — это область памяти, где данные могут выделяться динамически во время выполнения программы. В отличие от стека, где данные удаляются автоматически после завершения функции, данные в куче остаются, пока не будут явно удалены.

Куча идеально подходит для хранения данных, которые должны существовать дольше времени выполнения функции, или для работы с большими объёмами данных. Однако работа с кучей требует тщательного управления: если объекты не удаляются, когда они больше не нужны, это может привести к утечке памяти, что, в свою очередь, может вызвать исчерпание доступной памяти.



Объекты могут содержать методы, а методы — локальные переменные. Эти локальные переменные хранятся в стеке потока, даже если сам объект, которому принадлежат методы, находится в куче.

Куча — это область памяти, используемая для динамического распределения во время выполнения программы. В отличие от стека, данные в куче могут существовать дольше, чем отдельные вызовы функций, а объёмы памяти, выделяемой в куче, обычно гораздо больше, чем в стеке.

Динамическое управление памятью подразумевает процесс выделения и освобождения памяти в куче во время работы программы. Когда программе требуется память для хранения данных, она может запросить у операционной системы блок памяти в куче, достаточно большой для этих данных.

Однако важно помнить, что, в отличие от стека, память в куче не освобождается автоматически. Когда данные больше не нужны, программа должна явно указать операционной системе, что эту

память можно освободить для использования другими процессами. Этот процесс называется “освобождение памяти”.

Если программа продолжает использовать память, не освобождая её, это может привести к “утечке памяти”, когда значительные объёмы памяти заняты данными, которые уже не нужны. Это может вызвать снижение производительности и, в конечном итоге, привести к ошибкам, когда вся доступная память будет исчерпана.

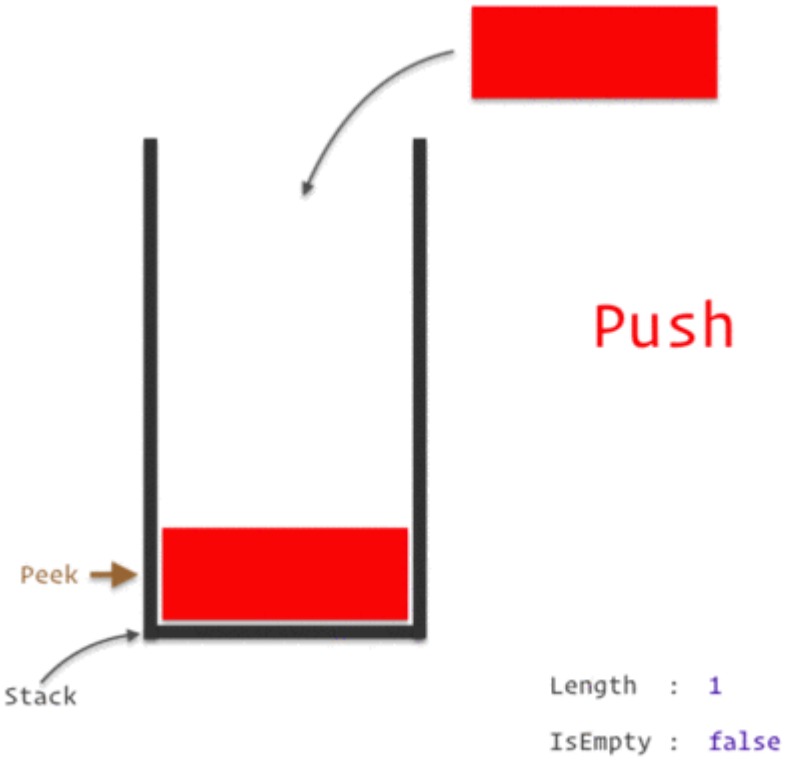
Особенности

Динамическое распределение памяти: Куча позволяет программе запрашивать необходимый объём памяти динамически и использовать его до тех пор, пока программа не освободит его.
Долговечность данных: Память в куче не освобождается автоматически, поэтому данные могут существовать до тех пор, пока не будут явно удалены, что позволяет им пережить вызовы функций.
Управление памятью: Работа с кучей требует тщательного управления памятью. Утечки памяти могут стать проблемой, если память не освобождается своевременно.
Медленный доступ: Доступ к данным в куче может быть медленнее по сравнению со стеком, из-за более сложного управления и отсутствия локализации данных.

Стек (Stack)

Стек — это область памяти, в которой функции хранят свои переменные и информацию, необходимую для выполнения.

Представьте стек как стопку подносов в ресторане: вы можете добавить поднос сверху (push) или взять верхний поднос (pop). Точно так же, когда вызывается функция, её локальные переменные и информация о вызове помещаются на вершину стека, а при завершении функции эти данные удаляются с вершины.



Стек обеспечивает быстрый доступ к данным и автоматическое управление памятью, но его размер ограничен. Если программа потребляет больше стековой памяти, чем доступно, это может привести к ошибке переполнения стека.

Когда вызывается функция, для неё выделяется блок памяти на вершине стека. Этот блок, известный как “фрейм стека”, содержит пространство для всех локальных переменных функции, а также информацию, такую как адрес возврата — место в коде, куда программа должна вернуться после завершения функции.

Когда одна функция вызывает другую, для новой функции выделяется собственный фрейм стека, и она становится текущей активной функцией. По завершении работы функции её фрейм стека удаляется, и управление передаётся обратно вызывающей функции.

Особенности работы

LIFO (Last-In, First-Out): Стек работает по принципу “последним пришёл — первым ушёл”. Это означает, что последняя вызванная функция будет первой, которая завершит работу и вернёт управление.
Автоматическое управление памятью: Когда функция завершает выполнение, её локальные переменные автоматически удаляются. Это упрощает управление памятью, так как нет необходимости в явном освобождении памяти.
Ограниченный размер: Размер стека обычно ограничен. Если программа попытается использовать больше памяти, чем доступно в стеке, это приведёт к ошибке переполнения стека.
Быстрый доступ: Доступ к данным в стеке обычно быстрее, чем к данным в куче, поскольку стек локализован в памяти, и данные из него могут загружаться в кэш процессора для ускоренного доступа.

Стек и куча в Java

Основное отличие стека и кучи в Java от их общего представления связано с автоматическим управлением памятью. В Java не нужно явно освобождать память в куче, так как этим занимается сборщик мусора. Если на объект, на который ссылается локальная переменная, больше нет ссылок, он становится доступным для сборщика мусора.

Пространство стека ограничено и обычно меньше пространства кучи. Если приложение попытается использовать больше стековой памяти, чем доступно, это приведёт к ошибке StackOverflowError.

Куча в Java — это область памяти, где создаются все объекты. Когда вы создаёте объект с помощью оператора new, он размещается в куче.

Сборщик мусора

Сборщик мусора (Garbage Collector, GC) — это процесс в виртуальной машине Java (JVM), который автоматически освобождает память, выделенную для объектов, которые больше не используются. Этот процесс происходит следующим образом:

Маркировка: Сборщик мусора начинает “маркировать” все объекты в куче, которые больше не доступны из корней приложения. “Корни” обычно включают в себя стек вызова и глобальные ссылки. Если на объект в куче не существует активной ссылки из этих “корней”, он считается недоступным.
Удаление: После того как недоступные объекты были помечены, сборщик мусора освобождает память, которую они занимали.

Несмотря на автоматизацию управления памятью, неэффективное использование ресурсов может по-прежнему привести к проблемам. Например, если приложение постоянно создаёт новые объекты и сохраняет на них ссылки, это может привести к утечке памяти, когда куча постепенно заполняется, и система больше не может выделить память для новых объектов. В таких случаях приложение может завершиться с ошибкой OutOfMemoryError.

```
public class Main {
    public static void main(String[] args) {
        Person person1 = new Person("John");
        Person person2 = new Person("Jane");

        System.out.println("Name of person1: " + person1.getName());
        System.out.println("Name of person2: " + person2.getName());

        swapNames(person1, person2);

        System.out.println("After swapping:");
        System.out.println("Name of person1: " + person1.getName());
```

```
        System.out.println("Name of person2: " + person2.getName());
    }

    public static void swapNames(Person p1, Person p2) {
        String temp = p1.getName();
        p1.setName(p2.getName());
        p2.setName(temp);
    }
}
```

В этом примере мы создаём два объекта Person в методе main(). Эти объекты создаются в куче, а ссылки на них (person1 и person2) хранятся в стеке.

Когда вызывается метод swapNames(), происходит следующее:

Создание ссылок в стеке: Ссылки p1 и p2 создаются в стеке как копии ссылок person1 и person2. Они указывают на те же объекты Person, что и оригинальные ссылки. Это возможно благодаря тому, что в Java передача объектов в методы осуществляется по значению ссылок, что позволяет методам изменять состояние объектов, на которые эти ссылки указывают.

Работа с временной переменной: Внутри метода swapNames() создаётся переменная temp для временного хранения имени объекта Person, на который указывает p1. Так как строка — это объект, она хранится в куче, а ссылка на неё — в переменной temp, которая размещена в стеке.

Замена имен: Мы меняем имя объекта, на который указывает p1, на имя объекта p2, а затем меняем имя объекта p2 на значение из переменной temp. Это изменение затрагивает сами объекты, находящиеся в куче.

Жизненный цикл локальных переменных: Память для локальных переменных, включая temp, выделяется при входе в метод. Для метода создаётся новый фрейм стека, содержащий место для всех локальных переменных и параметров. Этот фрейм имеет фиксированный размер, и все локальные переменные “резервируют” своё место при вызове метода, независимо от того, когда именно они инициализируются в теле метода. В нашем случае память для переменной temp выделяется сразу при вызове метода swapNames(), хотя присвоение значения происходит позже, при выполнении строки String temp = p1.getName();.

Завершение работы метода: После завершения работы метода swapNames() локальная переменная temp исчезает, так как она хранилась в стеке, и её жизненный цикл ограничен временем выполнения метода. Однако объекты Person продолжают существовать в куче, пока на них есть ссылки, и они не будут удалены сборщиком мусора.

В программировании стек и куча выполняют разные, но одинаково важные функции. Стек используется для хранения информации о вызовах функций и их локальных переменных, а куча — для динамического выделения памяти.

В Java управление памятью становится особенно интересным благодаря автоматической сборке мусора, которая освобождает неиспользуемую память, и разделению на области Heap Space и Stack Space. Эти механизмы делают Java удобной для разработки, однако требуют внимательного подхода к управлению памятью, чтобы избежать проблем с производительностью.

Таким образом, управление памятью — это сложная, но важная часть работы с программным обеспечением. Понимание основ этих процессов помогает разработчикам создавать более эффективные и надёжные приложения.

Class Loaders. Виды, для чего нужны - кратко

Class Loaders — это часть JVM, отвечающая за загрузку классов в память. Они обеспечивают динамическую загрузку, безопасность (проверка байт-кода) и изоляцию (для веб-приложений). Загрузка следует принципу делегирования: сначала родительский loader, затем свой.

Виды:

- **Bootstrap Class Loader**: Загружает core-классы (java.lang.*) из rt.jar. Написан на native-коде.
- **Extension Class Loader**: Загружает расширения из jre/lib/ext.
- **Application (System) Class Loader**: Загружает классы из CLASSPATH (ваши .class или .jar).

Для чего: Позволяют загружать классы на лету, поддерживают плагины, веб-сервера (как Tomcat). Пример: `Class.forName("MyClass")` использует текущий loader. Ссылки: Oracle Docs — https://docs.oracle.com/javase/specs/jvms/se21/html/jvms-5.html.

Особое внимание: Объект класса Class представляет метаданные класса (имя, поля, методы). Он создается JVM при загрузке. `Class clazz = MyClass.class;` — это ссылка на метаданные. Используется в рефлексии (для динамического доступа). Пример: `clazz.getMethods()` возвращает методы. Это фундаментально для понимания, как Java работает с типами на runtime (динамическое связывание в ООП).

Class Loaders - подробно

Определение и назначение: Class Loaders (загрузчики классов) в Java — это часть JVM, отвечающая за динамическую загрузку классов в память во время выполнения программы. Они позволяют загружать .class файлы из различных источников (файловая система, сеть, JAR-файлы) и обеспечивают модульность, изоляцию и гибкость. Class Loaders поддерживают принцип инкапсуляции (скрывают детали загрузки) и полиморфизм (разные загрузчики для разных задач). Используются для плагинов, горячей замены кода, контейнеров (Tomcat), и динамической загрузки ресурсов.

Ключевые аспекты:

- **Иерархия**: Bootstrap (нативные классы, rt.jar), Platform (модули Java), Application (classpath).

- **Делегирование**: Дочерний загрузчик запрашивает родительский перед загрузкой.
- **Кастомные загрузчики**: Для загрузки классов из нестандартных источников (например, сети).
- **Практическое использование**: Плагины (например, в IDE), загрузка JAR в runtime, изоляция (web-приложения в Tomcat).

Плюсы/минусы:

- Плюсы: Гибкость, динамичность, изоляция (разные версии классов).
- Минусы: Сложность отладки, утечки памяти при неправильной очистке.

Связь с ООП/SOLID: Инкапсуляция (логика загрузки скрыта), DIP (зависимость от абстракций ClassLoader).

Ссылки:

- Oracle Docs (ClassLoader): https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/ClassLoader.html
- Baeldung (Custom ClassLoader): https://www.baeldung.com/java-classloaders
- GeeksforGeeks: https://www.geeksforgeeks.org/classloader-in-java/

Практическое использование

Я покажу пример кастомного ClassLoader для загрузки класса из внешнего .class файла (например, из папки). Сценарий: Плагинная система, где класс загружается динамически, создается экземпляр через рефлексию, и вызывается метод. Это демонстрирует:

- Загрузку .class файла.
- Использование рефлексии (Class.forName, clazz.getMethod).
- Практическое применение (плагины).

Сценарий

1. Есть интерфейс Plugin с методом doWork().
2. Плагин (PluginImpl) реализует его, но хранится как .class файл в папке.
3. Кастомный ClassLoader загружает PluginImpl.class, создает объект, вызывает метод.

Шаги для подготовки

1. Создайте интерфейс и реализацию плагина.
2. Скомпилируйте их (javac).
3. Поместите .class файл плагина в папку (например, plugins/).
4. Программа загрузит класс и выполнит doWork().

Код программы

Интерфейс Plugin

```
// Plugin.java
public interface Plugin {
    void doWork();
}
```

Реализация PluginImpl (отдельный файл, компилируется в plugins/PluginImpl.class)

```
// PluginImpl.java
public class PluginImpl implements Plugin {
    @Override
    public void doWork() {
        System.out.println("PluginImpl is doing work!");
    }
}
```

Кастомный ClassLoader и демонстрация

```
import java.io.*;
import java.lang.reflect.Method;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class CustomClassLoader extends ClassLoader {
    private final String classPath;

    public CustomClassLoader(String classPath, ClassLoader parent) {
        super(parent);
        this.classPath = classPath;
    }

    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        try {
            // Читаем .class файл
            Path filePath = Paths.get(classPath, name.replace('.', '/') +
".class");
            byte[] classBytes = Files.readAllBytes(filePath);

            // Определяем класс
            return defineClass(name, classBytes, 0, classBytes.length);
        } catch (IOException e) {
            throw new ClassNotFoundException("Cannot load class " + name, e);
```

```
        }
    }
}

class PluginLoader {
    public static void loadAndExecutePlugin(String classPath, String className) {
        try {
            // Создаем кастомный загрузчик
            CustomClassLoader loader = new CustomClassLoader(classPath,
PluginLoader.class.getClassLoader());

            // Загружаем класс
            Class<?> clazz = loader.loadClass(className);
            System.out.println("Loaded class: " + clazz.getName());

            // Проверяем, реализует ли класс Plugin
            if (Plugin.class.isAssignableFrom(clazz)) {
                // Создаем экземпляр
                Plugin plugin = (Plugin)
clazz.getDeclaredConstructor().newInstance();

                // Вызываем метод
                plugin.doWork();
            } else {
                System.out.println("Class " + className + " does not implement
Plugin");
            }
        } catch (ClassNotFoundException e) {
            System.err.println("Class not found: " + e.getMessage());
        } catch (Exception e) {
            System.err.println("Error executing plugin: " + e.getMessage());
            e.printStackTrace();
        }
    }
}

public class ClassLoaderDemo {
    public static void main(String[] args) {
        // Путь к папке с .class файлами
        String classPath = "plugins";
        String className = "PluginImpl";

        // Загрузка и выполнение плагина
```

```
        PluginLoader.loadAndExecutePlugin(classPath, className);
    }
}
```

Подготовка и запуск

1. **Скомпилируйте Plugin и PluginImpl**:

   ```
   javac Plugin.java PluginImpl.java
   ```

2. **Переместите PluginImpl.class** в папку `plugins/`:

   ```
   mkdir plugins
   mv PluginImpl.class plugins/
   ```

3. **Скомпилируйте и запустите основной код**:

   ```
   javac ClassLoaderDemo.java
   java ClassLoaderDemo
   ```

Ожидаемый вывод:

```
Loaded class: PluginImpl
PluginImpl is doing work!
```

Объяснение кода

1. **Plugin**: Интерфейс с методом doWork(). Обеспечивает контракт для плагинов (DIP из SOLID).
2. **PluginImpl**: Реализация, которая будет загружаться динамически. Хранится отдельно как .class.
3. **CustomClassLoader**:
   - Наследуется от ClassLoader, принимает parent (для делегирования).
   - Переопределяет findClass: читает .class файл (Files.readAllBytes), преобразует в класс через defineClass.
   - classPath — папка с .class файлами.
   - Почему так: defineClass — стандартный API для определения класса из байтов. Путь формируется по имени класса (package.name -> package/name.class).
4. **PluginLoader**:
   - loadAndExecutePlugin: Создает CustomClassLoader, загружает класс, проверяет реализацию Plugin (isAssignableFrom), создает экземпляр через рефлексию, вызывает doWork().
   - Рефлексия (clazz.getDeclaredConstructor().newInstance()) позволяет создавать объекты без явного импорта.
   - Обработка исключений (ClassNotFoundException, etc.) для надежности.

5. **ClassLoaderDemo**: Запускает процесс, указывая путь и имя класса.

Почему такая реализация:

- **Практичность**: Демонстрирует плагинную систему, как в IntelliJ/Eclipse (загрузка расширений).
- **Инкапсуляция**: CustomClassLoader скрывает детали загрузки (SRP).
- **Полиморфизм**: Plugin интерфейс позволяет работать с любыми реализациями.
- **Обработка ошибок**: Пойманы ClassNotFoundException, NoSuchMethodException, и другие для robustness.
- **Lazy loading**: Класс загружается только при вызове.

Почему ClassLoader важен:

- **Плагины**: Загружают новые классы без перекомпиляции (например, Maven плагины).
- **Контейнеры**: Tomcat использует отдельные ClassLoader'ы для каждого web-приложения, изолируя классы.
- **Hot deployment**: Замена классов в runtime (JRebel).
- **Модульность**: JPMS (Java Platform Module System) использует ClassLoader'ы для изоляции модулей.

Потенциальные улучшения:

- Добавить проверку байт-кода (ClassFormatError).
- Поддержка JAR (URLClassLoader для чтения из архивов).
- Очистка загрузчиков (для избежания memory leaks).
- Многопоточность: synchronized в findClass для thread-safety.

Ссылки для углубления:

- Oracle Reflection API: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/reflect/package-summary.html
- Baeldung Reflection: https://www.baeldung.com/java-reflection
- Medium (Custom ClassLoader): https://medium.com/@aarkay0106/java-classloaders-a-deep-dive-with-practical-examples-92e4949f7064
- Oracle JPMS: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/module/package-summary.html

Тестирование

1. Создайте другой плагин (PluginImpl2 implements Plugin), скомпилируйте, поместите в plugins/.
2. Измените className в main на "PluginImpl2", проверьте работу.

3. Попробуйте несуществующий класс — увидите ClassNotFoundException.

Альтернативный сценарий

Для большей реалистичности можно загрузить класс из сети:

- Используйте URLClassLoader (extends ClassLoader).
- Укажите URL (http://server/PluginImpl.class).
- Пример: new URLClassLoader(new URL[]{new URL("http://...")}).loadClass("PluginImpl").

Код для URLClassLoader:

```
import java.net.URL;
import java.net.URLClassLoader;

public class URLClassLoaderDemo {
    public static void main(String[] args) throws Exception {
        URL[] urls = {new URL("file:plugins/")};
        try (URLClassLoader loader = new URLClassLoader(urls)) {
            Class<?> clazz = loader.loadClass("PluginImpl");
            Plugin plugin = (Plugin) clazz.getDeclaredConstructor().newInstance();
            plugin.doWork();
        }
    }
}
```

Почему так: URLClassLoader проще для файлов/JAR, но менее гибкий (не читает raw байты). CustomClassLoader показывает низкоуровневую работу.

Совет: Запустите код в IDE, проверьте с разными .class файлами. Если нужны детали (например, загрузка JAR или thread-safety) — дайте знать!

Классы и объекты

Класс — blueprint (шаблон) для объектов, определяющий структуру и поведение. Объект — экземпляр класса, имеющий состояние и поведение. Это основа ООП (абстракция и инкапсуляция).

Пример кода:

```
public class Dog {  // Класс
    String name;    // Поле (состояние)

    void bark() {   // Метод (поведение)
        System.out.println("Woof!");
```

```
    }
}

Dog myDog = new Dog();  // Объект
myDog.name = "Buddy";
myDog.bark();
```

Здесь класс абстрагирует концепцию "собака", объект — конкретный экземпляр.

Структура класса

Класс состоит из:

- Заголовок: Модификаторы, имя, extends/implements.
- Поля: Состояния.
- Конструкторы: Инициализация.
- Методы: Поведение.
- Блоки инициализации.
- Вложенные классы.

Пример:

```
public class MyClass extends Parent implements Interface {
    // Поля
    private int field;

    // Конструктор
    public MyClass() {}

    // Метод
    public void method() {}

    // Блок
    { System.out.println("Init"); }
}
```

Поля

Поля — переменные класса, представляющие состояние объекта (инкапсуляция в ООП). Могут быть статическими (принадлежат классу) или экземплярными (принадлежат объекту). Объявляются в теле класса.

Пример:

```
class Person {
    String name;       // Экземплярное поле (состояние конкретного человека)
    static int count;  // Статическое поле (общее для всех объектов)
}
```

Создание: Вне методов, можно инициализировать сразу (`int x = 0;`) или в конструкторе.

Конструкторы (ключевое слово super и this)

Конструкторы — специальные методы для инициализации объектов. Имя совпадает с классом, нет возвращаемого типа. По умолчанию есть пустой.

- **this**: Ссылка на текущий объект. Используется для вызова другого конструктора (`this(5);`) или доступа к полям (`this.name = name;`).
- **super**: Вызов конструктора родителя. Должен быть первой строкой.

Пример:

```
class Parent {
    Parent() { System.out.println("Parent"); }
}

class Child extends Parent {
    Child() {
        super();  // Вызов родителя
        System.out.println("Child");
    }

    Child(int x) {
        this();   // Вызов другого конструктора
    }
}
```

Плюсы: Обеспечивает правильную инициализацию (наследование в ООП).

Методы (сигнатура методов, перегрузка методов)

Методы — поведение класса (полиморфизм в ООП). Сигнатура: модификаторы, тип возврата, имя, параметры (без тела).

Перегрузка: Методы с одним именем, но разными параметрами (количество, типы). Разрешается компилятором (статическое связывание).

Пример:

```
class Calculator {
    int add(int a, int b) { return a + b; }  // Сигнатура: add(int, int)

    double add(double a, double b) { return a + b; }  // Перегрузка
}
```

Статические и нестатические блоки инициализации

Блоки — код в фигурных скобках для инициализации.

- **Статические**: Выполняются при загрузке класса (один раз). Для статических полей.
- **Нестатические**: Выполняются при создании объекта (перед конструктором).

Пример:

```
class Init {
    static { System.out.println("Static block"); }  // Один раз

    { System.out.println("Instance block"); }       // При каждом new
}
```

Модификаторы

Модификаторы контролируют доступ и поведение (инкапсуляция).

static, final, abstract

- **static**: Принадлежит классу, не объекту. Для полей, методов, блоков. Пример: `static int count;` — общее.
- **final**: Неизменяемый. Для переменных (константы), методов (нельзя переопределить), классов (нельзя наследовать). Пример: `final int MAX = 10;`.
- **abstract**: Для абстрактных классов/методов. Метод без тела, класс нельзя инстанцировать. Поддерживает полиморфизм.

private, protected, public, package private

- **public**: Доступен везде.
- **protected**: В пакете и подклассах.
- **private**: Только в классе.
- **package-private** (default): Только в пакете.

Пример: `private int secret;` — инкапсуляция.

Варианты установки значений свойств объектов

Свойства — поля. Варианты:

- **Поля напрямую**: `obj.field = value;`. Плюсы: Простота. Минусы: Нарушает инкапсуляцию, небезопасно.
- **Конструктор**: Установка при создании. Плюсы: Обязательная инициализация, immutable-объекты. Минусы: Нельзя изменить после.
- **Сеттеры**: Методы как `setField(value)`. Плюсы: Валидация, контроль. Минусы: Может сделать объект mutable, overhead.

Пример: Предпочтительно сеттеры для инкапсуляции.

Абстрактные классы и интерфейсы (отличия, где что лучше использовать)

Оба для абстракции и полиморфизма.

- **Абстрактный класс**: Может иметь реализацию, поля, конструкторы. Наследуется (extends). Используйте, когда нужна общая реализация (is-a отношение).
- **Интерфейс**: Только сигнатуры (с Java 8 — default-методы). Реализуется (implements). Множественное. Используйте для контрактов (has-a, поведение).

Отличия: Абстрактный класс — для иерархии, интерфейс — для гибкости. Пример: Абстрактный `Animal` с методом `eat()`, интерфейс `Flyable` с `fly()`.

Изменяемые и неизменяемые объекты (примеры неизменяемых классов в java, как сделать класс неизменяемым)

- **Mutable**: Состояние меняется (ArrayList).
- **Immutable**: Не меняется после создания (String, Integer). Плюсы: Thread-safe, простота. Минусы: Overhead на новые объекты.

Примеры immutable: String, BigInteger, все wrappers (Integer и т.д.).

Как сделать: Финальные поля, нет сеттеров, приватные конструкторы, глубокое копирование в геттерах.

Пример:

```
final class Immutable {
    private final int value;
```

```
    public Immutable(int value) { this.value = value; }
    public int getValue() { return value; }
}
```

Inner и Nested классы

- **Nested (статические)**: Вложенный класс, статический. Доступ к статическим членам внешнего.
- **Inner (нестатические)**: Вложенный, имеет ссылку на внешний объект. Доступ к всем членам.

Пример:

```
class Outer {
    static class Nested { }  // Без new Outer()
    class Inner { }          // Нужно new Outer().new Inner()
}
```

Используют для группировки (инкапсуляция).

Локальные и анонимные классы

- **Локальные**: Класс внутри метода. Виден только там. Для временной логики.
- **Анонимные**: Без имени, для одноразового использования (часто с интерфейсами). Пример: `Runnable r = new Runnable() { public void run() {} };`

Используют для полиморфизма без лишних классов.

Класс Object

Базовый класс для всех (java.lang.Object). Поддерживает наследование. Каждый класс implicitly extends Object.

Методы класса Object

Перечисляю все с описаниями:

- public String toString() — Возвращает строковое представление объекта (по умолчанию class@hash).
- public boolean equals(Object obj) — Сравнивает на равенство (по умолчанию ==).
- public int hashCode() — Возвращает хэш-код для коллекций.
- protected void finalize() — Вызывается перед GC (устарел).
- public Class<?> getClass() — Возвращает Class-объект.
- protected Object clone() throws CloneNotSupportedException — Копирует объект.

- public void notify() — Будит один поток.
- public void notifyAll() — Будит все потоки.
- public void wait() throws InterruptedException — Ждет уведомления.
- public void wait(long timeout) throws InterruptedException — Ждет с таймаутом.
- public void wait(long timeout, int nanos) throws InterruptedException — Ждет с таймаутом в наносекундах.

Контракт equals - hashCode

Контракт: Если equals=true, то hashCode одинаковый. Если hashCode разный, equals=false. Для коллекций (HashMap). Переопределяйте оба.

Пример:

```
@Override
public boolean equals(Object o) { /* логика */ }

@Override
public int hashCode() { return Objects.hash(field); }
```

Реализация equals() и hashCode() в Java

Определение и назначение: Методы `equals()` и `hashCode()` определены в классе `Object` и используются для сравнения объектов и их размещения в коллекциях (HashMap, HashSet). `equals()` проверяет равенство объектов по содержимому, `hashCode()` возвращает целочисленный хэш для эффективного хранения/поиска. Они связаны контрактом: если два объекта равны по `equals()`, их `hashCode()` должен быть одинаков. Используются для:

- Сравнения объектов (например, в `if (obj1.equals(obj2))`).
- Работы с хэш-коллекциями (HashSet, HashMap), где `hashCode()` определяет бакет, а `equals()` разрешает коллизии.

Связь с ООП/SOLID:

- **Полиморфизм**: Переопределение методов для кастомной логики.
- **Инкапсуляция**: Логика сравнения скрыта в классе.
- **SRP**: `equals()` и `hashCode()` отвечают за сравнение и хэширование.

Контракт:

1. **Рефлексивность**: `x.equals(x)` всегда true.
2. **Симметричность**: Если `x.equals(y)`, то `y.equals(x)`.
3. **Транзитивность**: Если `x.equals(y)` и `y.equals(z)`, то `x.equals(z)`.

4. **Консистентность**: `x.equals(y)` возвращает одинаковый результат при неизменных данных.
5. **Null**: `x.equals(null)` всегда false.
6. **hashCode контракт**:
   - Если `x.equals(y)`, то `x.hashCode() == y.hashCode()`.
   - `hashCode()` должен быть консистентным (одинаковый при неизменных полях).
   - Необязательно, но желательно: разные объекты — разные хэши (минимизация коллизий).

Плюсы/минусы:

- Плюсы: Корректная работа коллекций, точное сравнение.
- Минусы: Ошибки в реализации приводят к багам (дубли в HashSet, пропущенные ключи в HashMap).

Практическое использование:

- **HashSet/HashMap**: Без переопределения объекты сравниваются по ссылке (`==`), что некорректно для пользовательских классов.
- **Сравнение объектов**: Например, проверка двух пользователей по ID и имени.

Ссылки:

- Oracle Docs (Object): https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Object.html
- Baeldung (equals/hashCode): https://www.baeldung.com/java-equals-hashcode-contracts
- GeeksforGeeks: https://www.geeksforgeeks.org/equals-hashcode-methods-java/

Практический пример

Я создам класс `Student` с полями `id` (int) и `name` (String). Реализуем `equals()` и `hashCode()` для сравнения студентов по этим полям. Покажем, как это влияет на HashSet/HashMap, и продемонстрируем ошибки при неправильной реализации. Код включает тесты для проверки контракта.

Реализация

```
import java.util.*;

class Student {
    private final int id;
    private final String name;

    public Student(int id, String name) {
        this.id = id;
        this.name = name;
```

```
    }

    public int getId() { return id; }
    public String getName() { return name; }

    @Override
    public boolean equals(Object obj) {
        // Рефлексивность
        if (this == obj) {
            return true;
        }
        // Null и тип
        if (obj == null || getClass() != obj.getClass()) {
            return false;
        }
        // Приведение и сравнение полей
        Student other = (Student) obj;
        return id == other.id && Objects.equals(name, other.name);
    }

    @Override
    public int hashCode() {
        // Используем Objects.hash для консистентности
        return Objects.hash(id, name);
    }

    @Override
    public String toString() {
        return "Student{id=" + id + ", name='" + name + "'}";
    }
}

// Плохая реализация для демонстрации ошибок
class BadStudent {
    private final int id;
    private final String name;

    public BadStudent(int id, String name) {
        this.id = id;
        this.name = name;
    }

    // Плохой equals: только id
```

```java
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        BadStudent other = (BadStudent) obj;
        return id == other.id; // Игнорируем name
    }

    // Плохой hashCode: нарушает контракт
    @Override
    public int hashCode() {
        return id; // Только id, не учитывает name
    }

    @Override
    public String toString() {
        return "BadStudent{id=" + id + ", name='" + name + "'}";
    }
}

public class EqualsHashCodeDemo {
    public static void main(String[] args) {
        // Тест корректной реализации
        System.out.println("=== Testing Correct Student ===");
        Student s1 = new Student(1, "Alice");
        Student s2 = new Student(1, "Alice");
        Student s3 = new Student(2, "Bob");

        // Проверка equals
        System.out.println("s1.equals(s2): " + s1.equals(s2)); // true
        System.out.println("s1.equals(s3): " + s1.equals(s3)); // false
        System.out.println("s1.equals(null): " + s1.equals(null)); // false

        // Проверка hashCode
        System.out.println("s1.hashCode(): " + s1.hashCode());
        System.out.println("s2.hashCode(): " + s2.hashCode()); // Одинаковый
        System.out.println("s3.hashCode(): " + s3.hashCode()); // Разный

        // Тест с HashSet
        Set<Student> set = new HashSet<>();
        set.add(s1);
        set.add(s2); // Не добавится, т.к. equals=true
        set.add(s3);
```

```
        System.out.println("HashSet size: " + set.size()); // 2
        System.out.println("HashSet: " + set);

        // Тест с HashMap
        Map<Student, String> map = new HashMap<>();
        map.put(s1, "Student1");
        map.put(s2, "Student2"); // Перезапишет, т.к. equals=true
        map.put(s3, "Student3");
        System.out.println("HashMap: " + map);

        // Тест плохой реализации
        System.out.println("\n=== Testing BadStudent ===");
        BadStudent bs1 = new BadStudent(1, "Alice");
        BadStudent bs2 = new BadStudent(1, "Bob"); // Разные name, но id одинаковый

        // Проверка equals
        System.out.println("bs1.equals(bs2): " + bs1.equals(bs2)); // true (ошибка)

        // Проверка hashCode
        System.out.println("bs1.hashCode(): " + bs1.hashCode());
        System.out.println("bs2.hashCode(): " + bs2.hashCode()); // Одинаковый
(ошибка)

        // Тест с HashSet
        Set<BadStudent> badSet = new HashSet<>();
        badSet.add(bs1);
        badSet.add(bs2); // Не добавится, хотя name разный
        System.out.println("Bad HashSet size: " + badSet.size()); // 1 (ошибка)
        System.out.println("Bad HashSet: " + badSet);
    }
}
```

Объяснение кода

1. **Student**:
   - Поля: `id` (int), `name` (String). Immutable (final) для консистентности.
   - **equals**:
     - Проверяет рефлексивность (`this == obj`).
     - Проверяет null и тип (`getClass()` для точного соответствия).
     - Сравнивает поля: `id` напрямую, `name` через `Objects.equals` (null-safe).
   - **hashCode**: Использует `Objects.hash(id, name)` для генерации хэша на основе обоих полей. Это консистентно с equals().

- **toString**: Для читаемого вывода.

2. **BadStudent**:
   - Поля те же, но:
   - **equals**: Сравнивает только `id`, игнорируя `name`. Это нарушает контракт, так как объекты с одинаковым `id`, но разным `name`, считаются равными.
   - **hashCode**: Возвращает только `id`, игнорируя `name`. Это нарушает контракт: равные по `equals` объекты имеют одинаковый хэш, но разные по содержимому объекты тоже (ошибка).
3. **EqualsHashCodeDemo**:
   - **Тест Student**:
     - Проверяет `equals()`: s1 и s2 равны (true), s1 и s3 разные (false), null false.
     - Проверяет `hashCode()`: s1 и s2 одинаковые, s3 разный.
     - HashSet: s2 не добавляется (equals=true), размер 2.
     - HashMap: s2 перезаписывает s1 (equals=true).
   - **Тест BadStudent**:
     - `equals()` ошибочно считает bs1 и bs2 равными (из-за `name`).
     - HashSet добавляет только один объект, хотя name разные (размер 1).
     - Показывает, как плохая реализация ломает коллекции.

Ожидаемый вывод:

```
=== Testing Correct Student ===
s1.equals(s2): true
s1.equals(s3): false
s1.equals(null): false
s1.hashCode(): <some value>
s2.hashCode(): <same value>
s3.hashCode(): <different value>
HashSet size: 2
HashSet: [Student{id=2, name='Bob'}, Student{id=1, name='Alice'}]
HashMap: {Student{id=2, name='Bob'}=Student3, Student{id=1, name='Alice'}=Student2}

=== Testing BadStudent ===
bs1.equals(bs2): true
bs1.hashCode(): 1
bs2.hashCode(): 1
Bad HashSet size: 1
Bad HashSet: [BadStudent{id=1, name='Alice'}]
```

Почему такая реализация

- **Корректность**:
  - `Student` соблюдает контракт: `equals()` учитывает все поля, `hashCode()` согласован.
  - `Objects.equals` для `name` обрабатывает null.
  - `Objects.hash` генерирует качественный хэш, минимизируя коллизии.
- **Ошибки в BadStudent**:
  - Игнорирование `name` в `equals()` приводит к ложному равенству.
  - `hashCode()` на основе только `id` делает разные объекты неразличимыми в коллекциях.
  - Это показывает, почему важно учитывать все значимые поля.
- **Практичность**:
  - `Student` работает корректно в HashSet/HashMap, избегая дубликатов или пропусков.
  - Тесты демонстрируют реальные последствия (Set/Map behavior).
- **Альтернативы**:
  - Генерация через IDE (IntelliJ: Alt+Insert → equals() and hashCode()).
  - Apache Commons Lang (EqualsBuilder/HashCodeBuilder).
  - Java 14+ Records автоматически генерируют корректные `equals()`/`hashCode()`.

Потенциальные улучшения:

- Добавить валидацию (например, `name != null` в конструкторе).
- Для сложных классов использовать `@EqualsAndHashCode` (Lombok).
- Тестировать с большим количеством объектов для проверки коллизий.

Практические советы:

- Всегда переопределяйте `equals()` и `hashCode()` вместе.
- Учитывайте все поля, влияющие на "логическое равенство".
- Используйте `Objects` для null-safety.
- Тестируйте с коллекциями (HashSet/Map) для проверки.

Тестирование:

- Попробуйте добавить в HashSet объекты с одинаковым `id`, но разным `name`.
- Измените `BadStudent`, добавив `name` в `equals()` и `hashCode()`, проверьте исправление.
- Используйте дебаггер для анализа хэшей и бакетов.

Если нужны дополнительные примеры (например, с более сложным классом или коллекциями) или разбор конкретных багов — дайте знать!

Метод clone

Копирует объект (shallow по умолчанию). Реализуйте Cloneable, переопределите clone().

Пример:

```
class MyClone implements Cloneable {
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

Принципы ООП

- **Абстракция**: Скрытие деталей (абстрактные классы, интерфейсы).
- **Инкапсуляция**: Объединение данных и методов, контроль доступа (private поля, геттеры/сеттеры).
- **Наследование**: Переиспользование кода (extends).
- **Полиморфизм**: Один интерфейс, разные реализации (перегрузка, переопределение).

В темах выше: Классы — все принципы; Модификаторы — инкапсуляция; Наследование — в super.

Наследование и Ассоциация (определение, плюсы-минусы каждого)

- **Наследование**: Подкласс inherits от суперкласса (is-a). Плюсы: Переиспользование, полиморфизм. Минусы: Tight coupling, хрупкость (изменения в родителе ломают детей).
- **Ассоциация**: Объекты связаны (has-a, composition/aggregation). Плюсы: Гибкость, loose coupling. Минусы: Больше кода.

Пример: Car extends Vehicle (наследование); Car has Engine (ассоциация).

Переопределение методов

Изменение поведения в подклассе (полиморфизм). Сигнатура та же, @Override.

Пример:

```
class Parent { void method() {} }
class Child extends Parent { @Override void method() {} }
```

Статическое и динамическое связывание

- **Статическое**: Компилятор решает (перегрузка, статические методы).
- **Динамическое**: Runtime решает (переопределение, виртуальные методы).

Пример: Перегруженный метод — статическое; переопределенный — динамическое.

Статическое и динамическое связывание в Java

Определение и назначение:
Связывание (binding) в Java — это процесс, при котором JVM определяет, какой код или метод будет выполнен для конкретного вызова. Оно делится на **статическое** (compile-time) и **динамическое** (runtime) связывание. Эти концепции связаны с принципами ООП, особенно с полиморфизмом, и влияют на то, как методы и переменные выбираются в программе.

- **Статическое связывание**: Происходит на этапе компиляции. JVM определяет метод или переменную на основе типа ссылки (не объекта). Используется для:
  - Статических методов (`static`).
  - Финальных методов (`final`).
  - Приватных методов (`private`).
  - Переменных (полей).
  - Перегруженных методов (method overloading).
- **Динамическое связывание**: Происходит во время выполнения. JVM определяет метод на основе фактического типа объекта (не ссылки). Используется для:
  - Переопределенных методов (method overriding) в полиморфизме.
  - Виртуальных методов (все не-static, не-final, не-private методы).

Связь с ООП/SOLID:

- **Полиморфизм**: Динамическое связывание реализует runtime-полиморфизм (переопределение методов).
- **Инкапсуляция**: Статическое связывание (private методы) скрывает реализацию.
- **OCP (Open/Closed Principle)**: Динамическое связывание позволяет расширять поведение через подклассы.

Плюсы/минусы:

- **Статическое**:
  - Плюсы: Быстрее (решение на этапе компиляции), оптимизация компилятором.
  - Минусы: Менее гибкое, не поддерживает runtime-полиморфизм.
- **Динамическое**:
  - Плюсы: Гибкость, поддержка полиморфизма.

- Минусы: Небольшой overhead (JVM смотрит vtable).

Когда использовать:

- Статическое: Для утилитных методов, констант, оптимизации.
- Динамическое: Для полиморфных иерархий, расширяемости.

Ссылки:

- Oracle Docs (Polymorphism): https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html
- Baeldung (Static vs Dynamic Binding): https://www.baeldung.com/java-static-dynamic-binding
- GeeksforGeeks: https://www.geeksforgeeks.org/static-vs-dynamic-binding-in-java/

Практический пример

Я создам пример, демонстрирующий оба типа связывания:

- **Статическое**: Перегруженные методы, статические методы, поля.
- **Динамическое**: Переопределенные методы в иерархии классов.

Сценарий: Система животных, где `Animal` — базовый класс, `Dog` и `Cat` — подклассы. Покажем, как вызываются методы и поля в разных контекстах.

Полный код

```
// Базовый класс
class Animal {
    // Поле для статического связывания
    public String type = "Generic Animal";

    // Статический метод (статическое связывание)
    public static void staticSound() {
        System.out.println("Animal makes a sound (static)");
    }

    // Приватный метод (статическое связывание)
    private void privateMethod() {
        System.out.println("Private method in Animal");
    }

    // Перегруженный метод (статическое связывание)
    public void makeSound() {
        System.out.println("Animal makes a generic sound");
    }

    public void makeSound(String prefix) {
```

```
        System.out.println(prefix + " Animal makes a sound");
    }

    // Виртуальный метод для переопределения (динамическое связывание)
    public void describe() {
        privateMethod();
        System.out.println("This is an animal: " + type);
    }
}

// Подкласс Dog
class Dog extends Animal {
    public String type = "Dog"; // Поле скрывает родительское (статическое связывание)

    @Override
    public void describe() {
        System.out.println("This is a dog: " + type);
    }

    // Перегруженный метод (статическое связывание)
    public void makeSound(String prefix, int times) {
        System.out.println(prefix + " Dog barks " + times + " times");
    }

    // Статический метод (не переопределяется, статическое связывание)
    public static void staticSound() {
        System.out.println("Dog barks (static)");
    }
}

// Подкласс Cat
class Cat extends Animal {
    public String type = "Cat";

    @Override
    public void describe() {
        System.out.println("This is a cat: " + type);
    }
}

public class BindingDemo {
    public static void main(String[] args) {
```

```
        // Статическое связывание
        System.out.println("=== Static Binding ===");
        Animal animal = new Animal();
        Dog dog = new Dog();
        Cat cat = new Cat();

        // Поля (статическое, по типу ссылки)
        System.out.println("animal.type: " + animal.type); // Generic Animal
        System.out.println("dog.type: " + dog.type); // Dog
        Animal dogAsAnimal = new Dog();
        System.out.println("dogAsAnimal.type: " + dogAsAnimal.type); // Generic
Animal (не Dog!)

        // Статические методы (по типу ссылки)
        Animal.staticSound(); // Animal makes a sound (static)
        Dog.staticSound(); // Dog barks (static)
        dogAsAnimal.staticSound(); // Animal makes a sound (static)

        // Перегруженные методы (статическое, по сигнатуре)
        animal.makeSound(); // Animal makes a generic sound
        animal.makeSound("Prefix"); // Prefix Animal makes a sound
        dog.makeSound("Woof", 3); // Woof Dog barks 3 times

        // Динамическое связывание
        System.out.println("\n=== Dynamic Binding ===");
        Animal animalRef = new Dog(); // Ссылка Animal, объект Dog
        animalRef.describe(); // This is a dog: Dog (по типу объекта)

        animalRef = new Cat();
        animalRef.describe(); // This is a cat: Cat (по типу объекта)

        // Список для демонстрации полиморфизма
        System.out.println("\n=== Polymorphism with List ===");
        List<Animal> animals = Arrays.asList(new Animal(), new Dog(), new Cat());
        for (Animal a : animals) {
            a.describe(); // Вызывает метод в зависимости от типа объекта
        }
    }
}
```

Ожидаемый вывод

```
=== Static Binding ===
animal.type: Generic Animal
```

```
dog.type: Dog
dogAsAnimal.type: Generic Animal
Animal makes a sound (static)
Dog barks (static)
Animal makes a sound (static)
Animal makes a generic sound
Prefix Animal makes a sound
Woof Dog barks 3 times

=== Dynamic Binding ===
This is a dog: Dog
This is a cat: Cat

=== Polymorphism with List ===
This is an animal: Generic Animal
This is a dog: Dog
This is a cat: Cat
```

Объяснение кода

1. **Класс Animal**:
   - Поле `type`: Доступно для всех экземпляров, используется для демонстрации статического связывания.
   - `staticSound()`: Статический метод, привязывается к типу ссылки (Animal или Dog).
   - `privateMethod()`: Приватный, всегда статическое связывание, вызывается только в Animal.
   - `makeSound()`: Перегружен (два варианта), выбор по сигнатуре на этапе компиляции.
   - `describe()`: Виртуальный метод, предназначен для переопределения.
2. **Классы Dog и Cat**:
   - Переопределяют `describe()` для полиморфизма (динамическое связывание).
   - Поле `type` скрывает родительское (field hiding, статическое).
   - Dog добавляет перегруженный `makeSound(String, int)`.
   - `staticSound()` в Dog — отдельный метод (статические не переопределяются).
3. **BindingDemo**:
   - **Статическое связывание**:
     - Поля: `dogAsAnimal.type` возвращает "Generic Animal", так как тип ссылки — Animal (компилятор смотрит на ссылку).
     - Статические методы: `dogAsAnimal.staticSound()` вызывает Animal's метод, а не Dog's.

        - Перегруженные методы: `makeSound()` выбирается по аргументам (например, `makeSound("Woof", 3)` вызывает Dog's метод).
    - **Динамическое связывание**:
        - `animalRef.describe()` вызывает метод Dog или Cat в зависимости от объекта, а не ссылки (Animal). JVM использует vtable для выбора.
        - Список животных показывает полиморфизм: каждый объект вызывает свой `describe()`.

Почему такая реализация

- **Практичность**: Показывает реальные сценарии — поля, статические/перегруженные методы (статическое), переопределенные методы (динамическое).
- **Четкость**:
    - Статическое: Поля и статические методы привязаны к типу ссылки, что очевидно в `dogAsAnimal.type`.
    - Динамическое: `describe()` демонстрирует полиморфизм, ключевой для ООП.
- **Тестирование контрастов**: Сравнение `Animal` и `Dog` показывает, как JVM решает, что вызывать.
- **Полиморфизм в цикле**: Список `List<Animal>` демонстрирует, как динамическое связывание работает в коллекциях.

Почему это важно:

- **Статическое**: Оптимизация (компилятор знает точный метод), используется в утилитах (Math.abs), константах.
- **Динамическое**: Основа расширяемости (например, Spring использует полиморфизм для бинов).

Потенциальные улучшения:

- Добавить рефлексию для анализа методов (Method.invoke).
- Показать final методы (тоже статическое связывание).
- Добавить интерфейс для большей абстракции.

Тестирование:

- Измените `type` в Dog на другое значение, проверьте `dogAsAnimal.type`.
- Добавьте новый подкласс (например, Bird), проверьте `describe()`.
- Попробуйте вызвать `makeSound()` с неверными аргументами (компилятор выдаст ошибку — статическое).

Ссылки для углубления:

- Oracle Method Resolution: https://docs.oracle.com/javase/specs/jls/se21/html/jls-15.html#jls-15.12
- Medium (Binding): https://medium.com/@aarkay0106/static-vs-dynamic-binding-in-java-a-comprehensive-guide-8b2c4f8b7a1f
- Javatpoint: https://www.javatpoint.com/static-binding-and-dynamic-binding

Если нужны дополнительные примеры (например, с интерфейсами или рефлексией) или разбор конкретных случаев — дайте знать!

Оболочки примитивных типов. Основное API. Особенности сравнения значений.

Wrappers: Для примитивов (int -> Integer) для использования в коллекциях (ООП-абстракция).

Основное API (для Integer, аналогично другим как Boolean, Double):

- public static Integer valueOf(int i) — Создает объект (кэширует -128..127).
- public int intValue() — Возвращает примитив.
- public static int parseInt(String s) — Парсит строку.
- public static String toString(int i) — В строку.
- public boolean equals(Object obj) — Сравнивает значения.
- public int hashCode() — Хэш.
- public int compareTo(Integer another) — Сравнение.

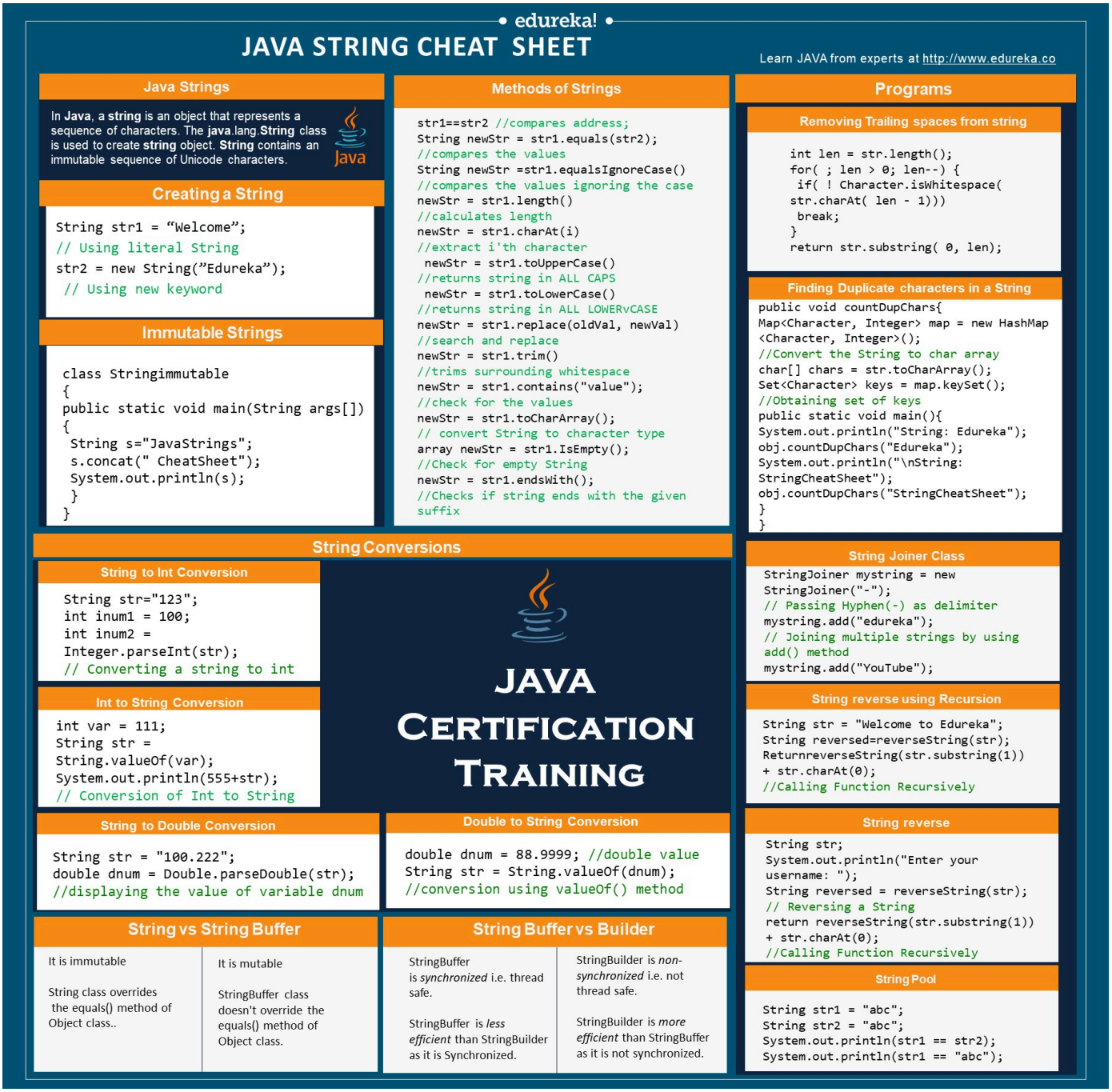
Сравнение: == для ссылок (кэшированные равны), equals для значений. Пример: Integer a = 100; Integer b = 100; a == b (true, кэш); для 1000 — false.

String

Immutable строка. Хранит char[].

Фишки работы со строками в Java

Строки (String) в Java — это один из самых часто используемых типов данных. Класс `String` находится в пакете `java.lang` и является immutable (неизменяемым), что обеспечивает безопасность и эффективность в многопоточной среде. Я подробно разберу все ключевые фишки: особенности класса, методы, String pool, StringBuilder и StringBuffer, трюки оптимизации, работу с регулярными выражениями. Затем перейду к анаграммам (как проверять, алгоритмы) и приведу примеры hard-задач с LeetCode, включая код решений и объяснения. Для визуализации вставлю диаграммы.



1. Основные особенности класса String

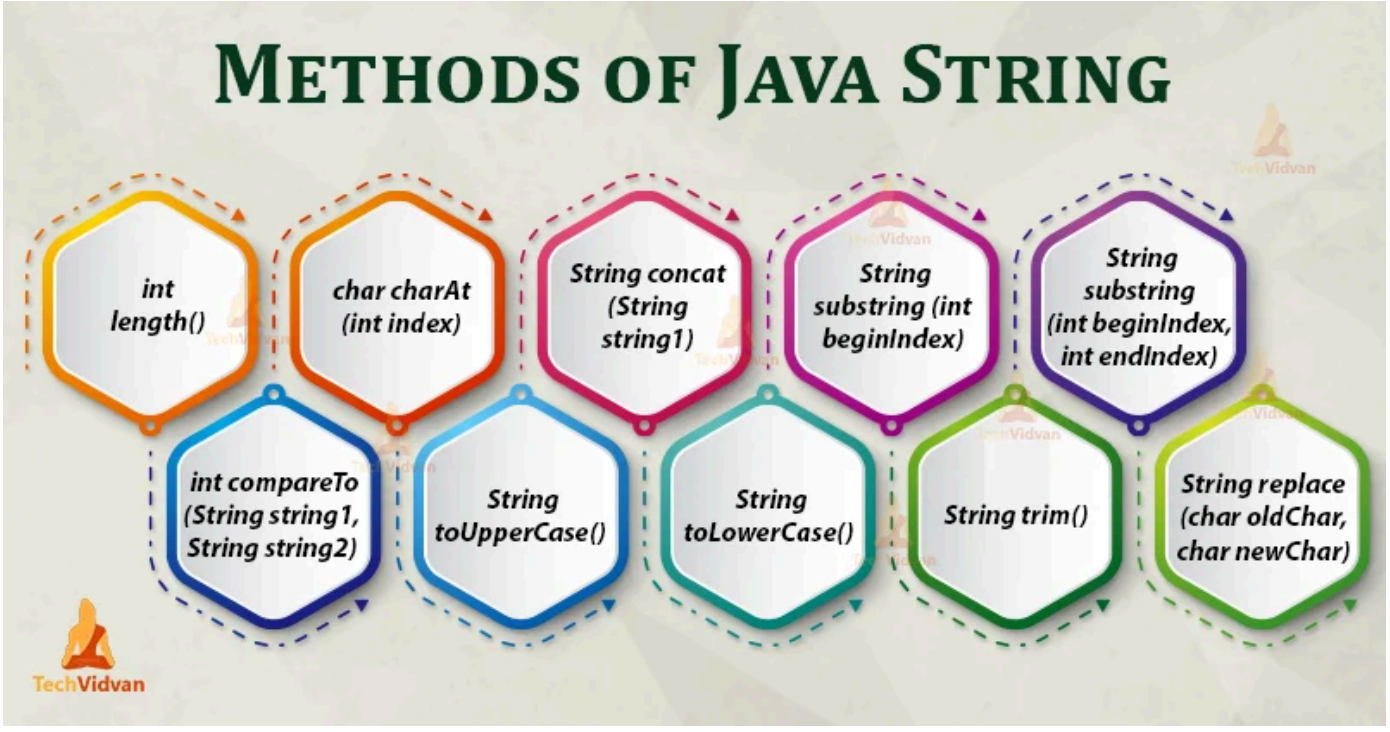
- **Immutable**: После создания строка не меняется. Любая операция (concat, replace) создает новую строку. Плюсы: Thread-safe, может использоваться в HashMap как ключ. Минусы: Overhead на память при частых изменениях.
- **String Pool**: JVM хранит уникальные строковые литералы в пуле (heap). "hello" == "hello" (true), но new String("hello") создает новый объект вне пула. Метод `intern()` возвращает ссылку из пула.
- **Хранение**: Внутренне — char[] (до Java 9) или byte[] (компактно для Latin-1).
- **Длина**: Ограничена Integer.MAX_VALUE (2^31 - 1).

Пример кода для демонстрации:

```
String s1 = "hello";  // В пуле
String s2 = "hello";  // Та же ссылка
```

```
String s3 = new String("hello");  // Новый объект

System.out.println(s1 == s2);  // true (ссылки одинаковые)
System.out.println(s1 == s3);  // false (разные объекты)
System.out.println(s1.equals(s3));  // true (содержимое одинаковое)
System.out.println(s3.intern() == s1);  // true (intern в пул)
```



2. Методы класса String (все основные с описаниями и примерами)

String имеет ~70 методов. Вот полный список категорий с примерами. Все методы immutable — возвращают новую строку.

- **Создание и базовые**:
  - `String()` — Пустая строка.
  - `String(char[] value)` — Из массива символов.
  - `String(String original)` — Копия.
  - `length()` — Длина строки. Пример: "hello".length() → 5.
- **Сравнение**:
  - `equals(Object anObject)` — Сравнивает содержимое (case-sensitive). Пример: "Hello".equals("hello") → false.
  - `equalsIgnoreCase(String anotherString)` — Игнорирует регистр. Пример: "Hello".equalsIgnoreCase("hello") → true.
  - `compareTo(String anotherString)` — Лексикографическое сравнение (возвращает int: отрицательное если this < another, 0 если равны, положительное если this > another). Пример: "a".compareTo("b") → -1.
  - `compareToIgnoreCase(String str)` — Аналогично, но без учета регистра.

- **Поиск и индексация**:
  - `charAt(int index)` — Символ по индексу (0-based). Пример: "hello".charAt(1) → 'e'.
  - `indexOf(int ch)` — Индекс первого вхождения символа. Пример: "hello".indexOf('l') → 2.
  - `indexOf(String str)` — Индекс подстроки. Пример: "hello".indexOf("lo") → 3.
  - `lastIndexOf(int ch)` — Последнее вхождение. Пример: "hello".lastIndexOf('l') → 3.
  - `contains(CharSequence s)` — Содержит ли подстроку. Пример: "hello".contains("ell") → true.
  - `startsWith(String prefix)` — Начинается ли с префикса. Пример: "hello".startsWith("he") → true.
  - `endsWith(String suffix)` — Заканчивается ли суффиксом. Пример: "hello".endsWith("lo") → true.
- **Манипуляции**:
  - `substring(int beginIndex)` — Подстрока от beginIndex. Пример: "hello".substring(1) → "ello".
  - `substring(int beginIndex, int endIndex)` — От begin до end (не включая end). Пример: "hello".substring(1, 3) → "el".
  - `concat(String str)` — Конкатенация. Пример: "hel".concat("lo") → "hello".
  - `replace(char oldChar, char newChar)` — Замена символа. Пример: "hello".replace('l', 'p') → "heppo".
  - `replace(CharSequence target, CharSequence replacement)` — Замена подстроки. Пример: "hello".replace("ll", "y") → "heyo".
  - `trim()` — Удаляет ведущие/завершающие пробелы. Пример: " hello ".trim() → "hello".
  - `toLowerCase()` — В нижний регистр. Пример: "Hello".toLowerCase() → "hello".
  - `toUpperCase()` — В верхний. Пример: "hello".toUpperCase() → "HELLO".
  - `split(String regex)` — Разделение по regex. Пример: "a,b,c".split(",") → ["a", "b", "c"].
  - `join(CharSequence delimiter, CharSequence... elements)` — Статический, соединяет строки. Пример: String.join("-", "a", "b") → "a-b".
  - `format(String format, Object... args)` — Форматирование. Пример: String.format("Name: %s, Age: %d", "Alice", 30) → "Name: Alice, Age: 30".
- **Проверки**:
  - `isEmpty()` — Пустая ли (length == 0). Пример: "".isEmpty() → true.
  - `isBlank()` (Java 11+) — Пустая или только пробелы. Пример: " ".isBlank() → true.
  - `matches(String regex)` — Соответствует regex. Пример: "123".matches("\d+") → true.
  - `codePointAt(int index)` — Unicode code point по индексу.
- **Преобразования**:
  - `toCharArray()` — В char[]. Пример: "hello".toCharArray() → ['h','e','l','l','o'].

- `valueOf(Object obj)` — Статический, в строку. Пример: String.valueOf(123) → "123".
- `getBytes()` — В byte[] (UTF-8 по умолчанию). Пример: "hello".getBytes().

- **Фишки производительности**:
  - Избегайте + в циклах (создает много объектов). Используйте StringBuilder.
  - String pool: Используйте литералы для экономии памяти.
  - Для больших строк: StringBuilder для mutable операций.
  - Regex: Pattern.compile для повторного использования (эффективнее matches).

Пример оптимизации:

```
// Плохо: В цикле +
String result = "";
for (int i = 0; i < 1000; i++) {
    result += i;
}  // Создает ~1000 строк

// Хорошо: StringBuilder
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 1000; i++) {
    sb.append(i);
}
String result = sb.toString();  // Одна строка
```

3. StringBuilder и StringBuffer

- **StringBuilder**: Mutable, не thread-safe, быстрее. Для однопоточных операций.
- **StringBuffer**: Mutable, thread-safe (synchronized), медленнее. Для многопоточки.
- Методы: append, insert, delete, reverse, replace, capacity, ensureCapacity, substring, toString.

Пример:

```
StringBuilder sb = new StringBuilder("hello");
sb.append(" world");  // "hello world"
sb.insert(5, ",");  // "hello, world"
sb.delete(5, 6);  // "hello world"
sb.reverse();  // "dlrow olleh"
System.out.println(sb);  // dlrow olleh
```

Трюк: Для реверса строки используйте sb.reverse().

4. Регулярные выражения (Regex) с String

Java использует `java.util.regex.Pattern` и `Matcher`.

- `matches(regex)`: Полное совпадение.

- `replaceAll(regex, replacement)`: Замена по regex.
- `split(regex)`: Разделение.

Пример:

```
import java.util.regex.*;

String text = "The quick brown fox jumps over the lazy dog.";
Pattern pattern = Pattern.compile("\\b\\w{4}\\b");  // Слова из 4 букв
Matcher matcher = pattern.matcher(text);
while (matcher.find()) {
    System.out.println(matcher.group());  // over, lazy
}

text = text.replaceAll("fox", "cat");  // Замена
System.out.println(text);  // The quick brown cat jumps over the lazy dog.
```

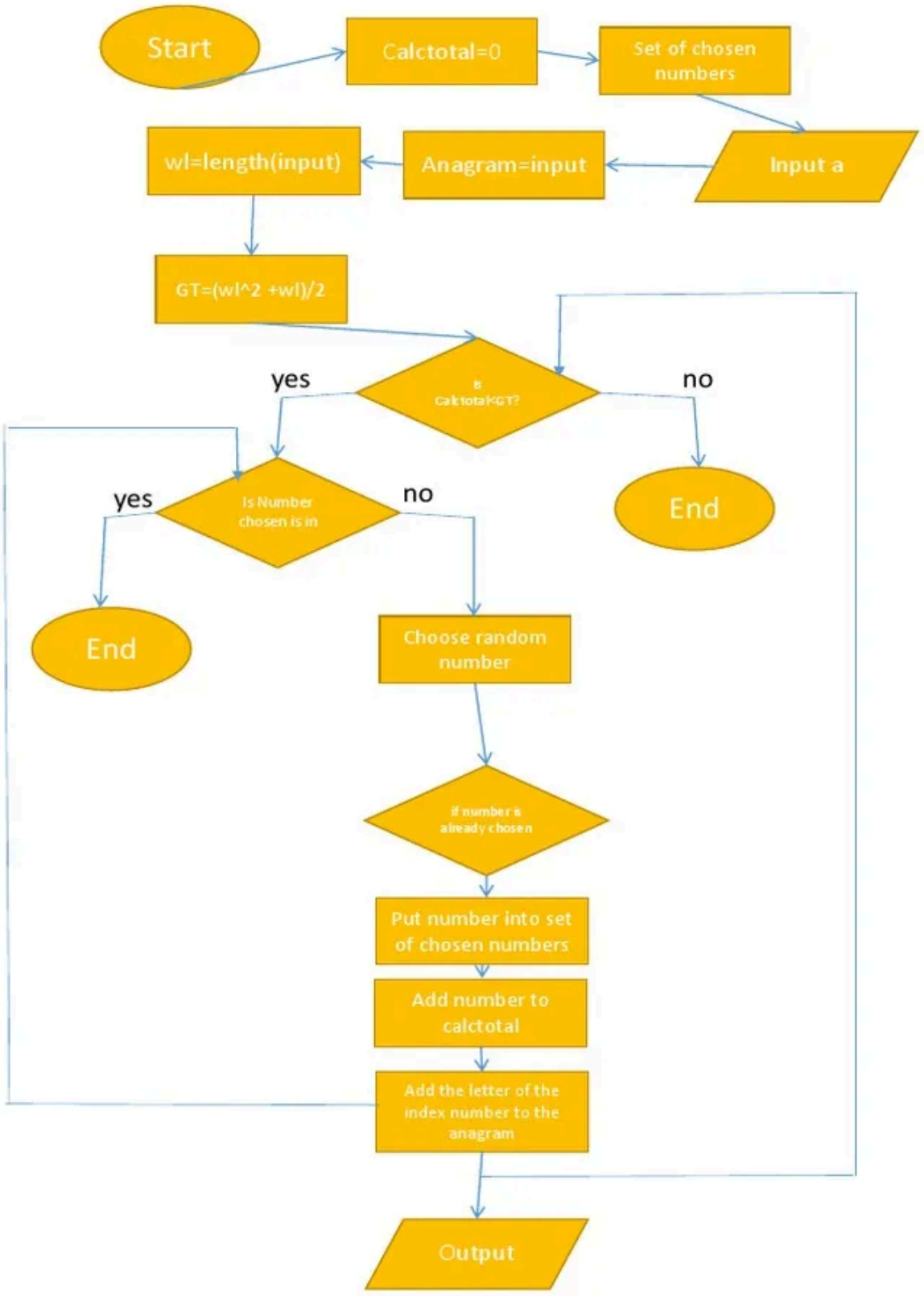
Фишки: Используйте `Pattern.compile` для повторных поисков (эффективно). Группы: ( ) для capture.

5. Анаграммы: Фишки и алгоритмы

Анаграмма — строки с одинаковыми символами в разном порядке (игнорируя регистр/пробелы).

Проверка анаграмм:

- **Сортировка**: Отсортировать символы, сравнить. O(n log n).
- **Счетчик символов**: Массив (char[26] для букв) или HashMap. O(n), space O(1) для фиксированного алфавита.
- Трюки: Игнорировать пробелы/регистр (toLowerCase, replaceAll("\s+", "")).
- Edge cases: Разные длины — не анаграммы, пустые строки — да.



Пример кода (счетчик):

```
public boolean isAnagram(String s1, String s2) {
    s1 = s1.replaceAll("\\s", "").toLowerCase();
```

```
    s2 = s2.replaceAll("\\s", "").toLowerCase();
    if (s1.length() != s2.length()) return false;

    int[] count = new int[26];
    for (char c : s1.toCharArray()) {
        count[c - 'a']++;
    }
    for (char c : s2.toCharArray()) {
        count[c - 'a']--;
        if (count[c - 'a'] < 0) return false;
    }
    return true;
}

// Тест
System.out.println(isAnagram("listen", "silent")); // true
System.out.println(isAnagram("hello", "world")); // false
```

Группировка анаграмм: Используйте HashMap<String, List>, ключ — отсортированная строка или сигнатура (char count string).

6. Примеры hard-задач с LeetCode

На основе поиска, вот 4 hard строковых проблемы. Я дам описание, код решения (Java) и объяснение. Эти задачи часто включают DP, sliding window, regex matching.

a. 76. Minimum Window Substring (Hard)

Описание: Найти минимальное окно в s, содержащее все символы t (с кратностями). Если нет — "".

Решение (Sliding Window):

```
public String minWindow(String s, String t) {
    if (s.length() < t.length()) return "";
    int[] count = new int[128];
    for (char c : t.toCharArray()) count[c]++;
    int left = 0, minLen = Integer.MAX_VALUE, minStart = 0, required = t.length();

    for (int right = 0; right < s.length(); right++) {
        if (count[s.charAt(right)]-- > 0) required--;
        while (required == 0) {
            if (right - left + 1 < minLen) {
                minLen = right - left + 1;
                minStart = left;
            }
```

```
            if (++count[s.charAt(left++)] > 0) required++;
        }
    }
    return minLen == Integer.MAX_VALUE ? "" : s.substring(minStart, minStart +
minLen);
}
```

Объяснение: Используем массив для подсчета нужных символов. Расширяем right, уменьшаем required. Когда required=0, сжимаем left, обновляем минимум. O(n) time.

b. 10. Regular Expression Matching (Hard)

Описание: Реализовать regex matching для '.' (любой char) и '*' (0+ предыдущих).

Решение (DP):

```
public boolean isMatch(String s, String p) {
    int m = s.length(), n = p.length();
    boolean[][] dp = new boolean[m+1][n+1];
    dp[0][0] = true;

    for (int j = 1; j <= n; j++) {
        if (p.charAt(j-1) == '*') dp[0][j] = dp[0][j-2];
    }

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (p.charAt(j-1) == '*') {
                dp[i][j] = dp[i][j-2] || (dp[i-1][j] && (s.charAt(i-1) ==
p.charAt(j-2) || p.charAt(j-2) == '.'));
            } else {
                dp[i][j] = dp[i-1][j-1] && (s.charAt(i-1) == p.charAt(j-1) ||
p.charAt(j-1) == '.');
            }
        }
    }
    return dp[m][n];
}
```

Объяснение: DP таблица dp[i][j] — совпадает ли s[0..i) с p[0..j). Обрабатываем '' *(0 или несколько)* и '.' *(любой)*. *O(mn)* time/space.

c. 32. Longest Valid Parentheses (Hard)

Описание: Найти длину самой длинной валидной подстроки скобок "()".

Решение (Stack):

```
public int longestValidParentheses(String s) {
    Stack<Integer> stack = new Stack<>();
    stack.push(-1);
    int max = 0;
    for (int i = 0; i < s.length(); i++) {
        if (s.charAt(i) == '(') {
            stack.push(i);
        } else {
            stack.pop();
            if (stack.isEmpty()) {
                stack.push(i);
            } else {
                max = Math.max(max, i - stack.peek());
            }
        }
    }
    return max;
}
```

Объяснение: Стек хранит индексы '('. При ')', pop. Если пусто — новый start. Вычисляем длину (i - peek). O(n) time.

d. 87. Scramble String (Hard)

Описание: Проверить, является ли s2 скрэмблом s1 (бинарное дерево перестановок).

Решение (Recursion + Memo):

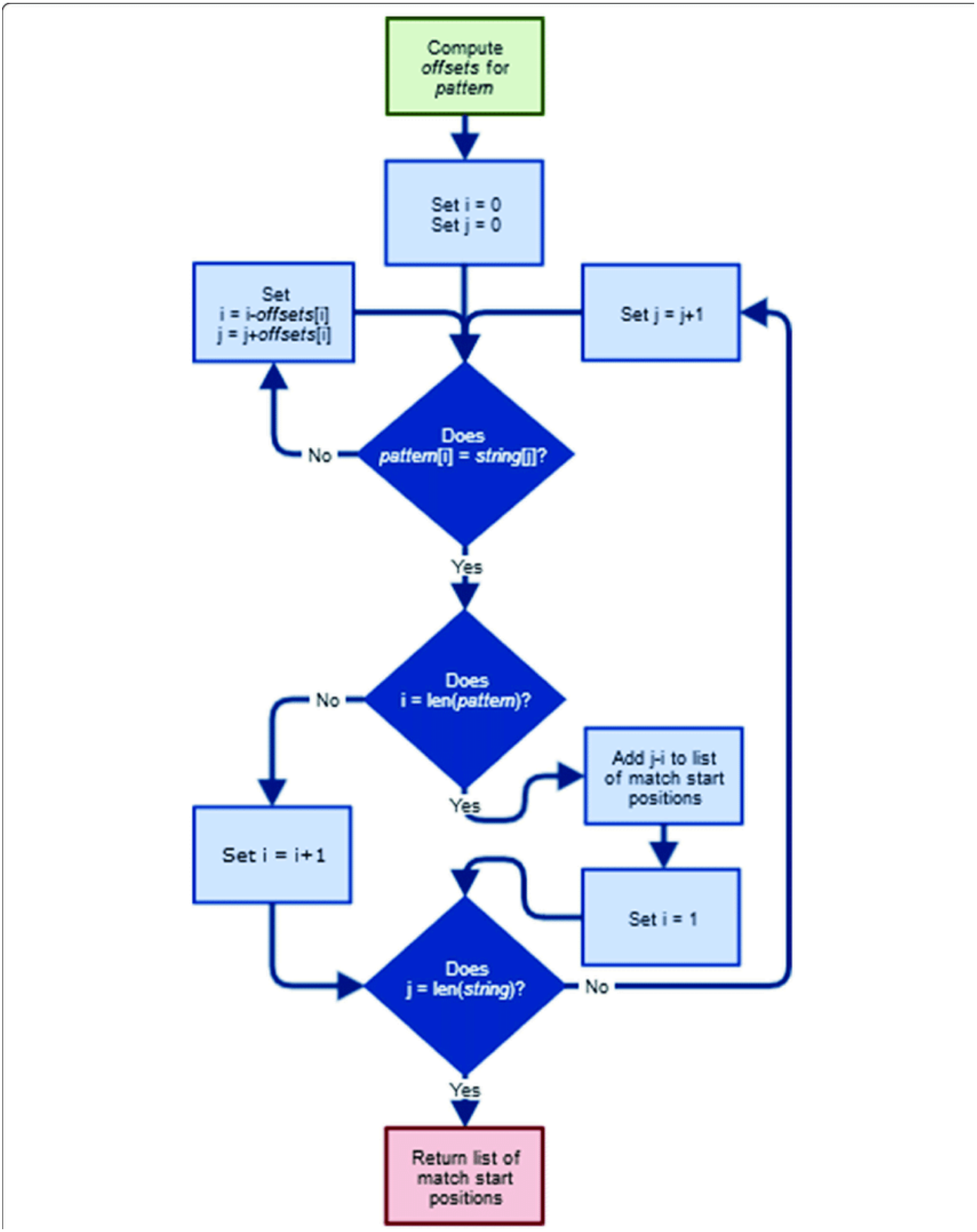
```
public boolean isScramble(String s1, String s2) {
    Map<String, Boolean> memo = new HashMap<>();
    return helper(s1, s2, memo);
}

private boolean helper(String s1, String s2, Map<String, Boolean> memo) {
    String key = s1 + "#" + s2;
    if (memo.containsKey(key)) return memo.get(key);
    if (s1.equals(s2)) return true;
    if (s1.length() != s2.length()) return false;

    int[] count = new int[26];
    for (char c : s1.toCharArray()) count[c - 'a']++;
    for (char c : s2.toCharArray()) if (--count[c - 'a'] < 0) return false;
```

```
    for (int i = 1; i < s1.length(); i++) {
        if (helper(s1.substring(0, i), s2.substring(0, i), memo) &&
helper(s1.substring(i), s2.substring(i), memo)) {
            memo.put(key, true);
            return true;
        }
        if (helper(s1.substring(0, i), s2.substring(s2.length() - i), memo) &&
helper(s1.substring(i), s2.substring(0, s2.length() - i), memo)) {
            memo.put(key, true);
            return true;
        }
    }
    memo.put(key, false);
    return false;
}
```

Объяснение: Рекурсия проверяет все разбиения, с memo для оптимизации. Сначала проверка частот. O(n^4) time worst, но memo помогает.



String pool

String Pool в Java — это область в heap (String Constant Pool), где хранятся уникальные строковые литералы для экономии памяти. Когда создается строка через литерал (`"hello"`), JVM проверяет пул: если строка есть, возвращается ссылка; если нет, строка добавляется. Метод `intern()` принудительно добавляет строку в пул.

Пример:

```
public class StringPoolDemo {
    public static void main(String[] args) {
        String s1 = "hello"; // В пуле
        String s2 = "hello"; // Та же ссылка из пула
        String s3 = new String("hello"); // Новый объект вне пула
        String s4 = s3.intern(); // Ссылка из пула

        System.out.println(s1 == s2); // true (одинаковые ссылки из пула)
        System.out.println(s1 == s3); // false (s3 - новый объект)
        System.out.println(s1 == s4); // true (s4 из пула)
        System.out.println(s1.equals(s3)); // true (содержимое одинаковое)
    }
}
```

Объяснение:

- `s1` и `s2` указывают на одну строку в пуле (экономия памяти).
- `new String("hello")` создает новый объект в heap, а не в пуле.
- `intern()` возвращает ссылку из пула, поэтому `s4 == s1`.
- `equals()` сравнивает содержимое, а `==` — ссылки.

Фишка: Используйте литералы или `intern()` для экономии памяти в коллекциях (HashMap/Set).
Ссылка: Oracle Docs — https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html#intern()

API (String API)

Перечисляю методы:

- public int length() — Длина строки.
- public char charAt(int index) — Символ по индексу.
- public String substring(int beginIndex) — Подстрока от beginIndex.
- public String substring(int beginIndex, int endIndex) — Подстрока от begin до end.
- public boolean equals(Object anObject) — Сравнение содержимого.
- public int compareTo(String anotherString) — Лексикографическое сравнение.
- public String toLowerCase() — В нижний регистр.
- public String toUpperCase() — В верхний регистр.
- public String trim() — Удаляет пробелы с краев.
- public boolean startsWith(String prefix) — Начинается с prefix.

- public boolean endsWith(String suffix) — Заканчивается на suffix.
- public int indexOf(String str) — Индекс первого вхождения.
- public int lastIndexOf(String str) — Индекс последнего вхождения.
- public String replace(char oldChar, char newChar) — Замена символа.
- public String replace(CharSequence target, CharSequence replacement) — Замена подстроки.
- public String[] split(String regex) — Разделение по regex.
- public static String valueOf(Object obj) — В строку.
- public boolean isEmpty() — Пустая ли.
- public String concat(String str) — Конкатенация.

И многие другие. Ссылки: Oracle Docs — https://docs.oracle.com/javase/21/docs/api/java/lang/String.html.

StringBuilder, StringBuffer

Для mutable строк. StringBuilder — не thread-safe, быстрее; StringBuffer — thread-safe (synchronized).

Примеры кода:

Создание/объявление:

```
StringBuilder sb = new StringBuilder();  // Пустой
StringBuilder sb2 = new StringBuilder("initial");  // С инициалом

StringBuffer sbf = new StringBuffer();  // Аналогично
```

Использование:

```
sb.append("hello").append(" world");  // Добавляет, возвращает себя (chain)
sb.insert(5, "beautiful ");           // Вставка
sb.delete(0, 5);                      // Удаление
sb.reverse();                         // Реверс
String result = sb.toString();        // В String
```

Для StringBuffer — то же, но synchronized для многопоточки.

Ссылки: Baeldung — https://www.baeldung.com/java-string-builder-buffer; Oracle — https://docs.oracle.com/javase/21/docs/api/java/lang/StringBuilder.html.

Подробное использование StringBuilder и StringBuffer в Java

StringBuilder и **StringBuffer** — это классы в Java, предназначенные для работы с изменяемыми (mutable) строками, в отличие от неизменяемого (immutable) класса `String`. Они используются для эффективной манипуляции строками, особенно при частых операциях конкатенации, вставки, удаления или замены. Я подробно разберу их особенности, различия, методы, сценарии применения, оптимизацию производительности, а также предоставлю примеры кода с объяснениями, включая реальные случаи и потенциальные подводные камни. Все будет оформлено в соответствии с вашим запросом, с акцентом на практическое использование и ссылки на источники.

---

1. Основные характеристики StringBuilder и StringBuffer

StringBuilder

- **Пакет**: `java.lang`
- **Назначение**: Предоставляет изменяемую последовательность символов для однопоточных приложений.
- **Thread-safety**: **Не thread-safe**. Методы не синхронизированы, что делает его быстрее.
- **Производительность**: Быстрее `StringBuffer` из-за отсутствия синхронизации.
- **Введен**: Java 5 (2004).
- **Когда использовать**: В однопоточных сценариях, где нужна высокая производительность (например, построение строк в цикле, парсинг данных).

StringBuffer

- **Пакет**: `java.lang`
- **Назначение**: То же, что StringBuilder, но для многопоточных приложений.
- **Thread-safety**: **Thread-safe**. Все методы синхронизированы (`synchronized`), что предотвращает data race.
- **Производительность**: Медленнее из-за накладных расходов на синхронизацию.
- **Введен**: Java 1.0 (1996).
- **Когда использовать**: В многопоточных приложениях, где несколько потоков могут изменять одну строку (например, логирование в сервере).

Общие черты

- **Mutable**: В отличие от `String`, изменяют содержимое без создания новых объектов.
- **Внутреннее хранение**: Массив символов (`char[]` до Java 9, `byte[]` после для компактности).
- **Авто-расширение**: Если массив заполняется, размер увеличивается (обычно удваивается).
- **Методы**: Похожи (append, insert, delete, etc.), но StringBuffer синхронизирован.
- **Связь с ООП/SOLID**:
  - **Инкапсуляция**: Внутренний массив скрыт, доступ через методы.

- **SRP**: Отвечают за манипуляцию строками.
- **Полиморфизм**: Реализуют интерфейсы `CharSequence`, `Appendable`.

Различия

Характеристика	StringBuilder	StringBuffer
Thread-safety	Не thread-safe	Thread-safe (synchronized)
Производительность	Быстрее	Медленнее
Введен	Java 5	Java 1.0
Использование	Однопоточные приложения	Многопоточные приложения

Примечание: В 99% случаев используйте `StringBuilder` (если не нужна многопоточность), так как он быстрее. StringBuffer редко нужен в современном Java благодаря `ConcurrentStringBuilder` или другим синхронизированным альтернативам.

Ссылки:

- Oracle Docs (StringBuilder): https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/StringBuilder.html
- Oracle Docs (StringBuffer): https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/StringBuffer.html
- Baeldung (StringBuilder vs StringBuffer): https://www.baeldung.com/java-string-builder-buffer

---

2. Основные методы StringBuilder и StringBuffer

Оба класса имеют идентичные методы (разница только в синхронизации StringBuffer). Вот полный список ключевых методов с описаниями и примерами:

- **Конструкторы**:
  - `StringBuilder()`: Пустой, начальная емкость 16.
  - `StringBuilder(int capacity)`: Указанная емкость.
  - `StringBuilder(String str)`: Инициализация строкой (+16 к длине).
  - `StringBuilder(CharSequence seq)`: Из CharSequence.
- **Манипуляции**:
  - `append(X x)`: Добавляет в конец (поддерживает int, double, String, char[], etc.). Возвращает this (fluent interface).
  - `insert(int offset, X x)`: Вставляет в позицию offset.
  - `delete(int start, int end)`: Удаляет [start, end).
  - `deleteCharAt(int index)`: Удаляет символ по индексу.

- `replace(int start, int end, String str)`: Заменяет [start, end) на str.
- `reverse()`: Реверсирует строку.

- **Информация**:
  - `length()`: Текущая длина.
  - `capacity()`: Размер внутреннего массива.
  - `charAt(int index)`: Символ по индексу.
  - `substring(int start)`: Подстрока от start.
  - `substring(int start, int end)`: Подстрока [start, end).
  - `indexOf(String str)`: Индекс первого вхождения.
  - `lastIndexOf(String str)`: Последнее вхождение.
- **Управление емкостью**:
  - `ensureCapacity(int minimumCapacity)`: Гарантирует минимальную емкость.
  - `trimToSize()`: Урезает массив до текущей длины.
- **Преобразование**:
  - `toString()`: В String.

Пример методов:

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World"); // Hello World
sb.insert(5, ","); // Hello, World
sb.delete(5, 6); // Hello World
sb.replace(0, 5, "Hi"); // Hi World
sb.reverse(); // dlroW iH
System.out.println(sb.length()); // 8
System.out.println(sb.capacity()); // 16 (или больше, если расширялся)
System.out.println(sb.toString()); // dlroW iH
```

---

3. Практические примеры использования

Пример 1: Построение строки в цикле (StringBuilder)

Сценарий: Генерация CSV-строки из списка чисел. Сравнение с `String` для производительности.

```
import java.util.*;

public class StringBuilderCsvDemo {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
```

```
        // Плохо: String конкатенация
        long startTime = System.nanoTime();
        String result = "";
        for (int num : numbers) {
            result += num + ",";
        }
        long stringTime = System.nanoTime() - startTime;

        // Хорошо: StringBuilder
        startTime = System.nanoTime();
        StringBuilder sb = new StringBuilder();
        for (int num : numbers) {
            sb.append(num).append(",");
        }
        String builderResult = sb.toString();
        long builderTime = System.nanoTime() - startTime;

        System.out.println("String result: " + result);
        System.out.println("StringBuilder result: " + builderResult);
        System.out.println("String time: " + stringTime + " ns");
        System.out.println("StringBuilder time: " + builderTime + " ns");
    }
}
```

Объяснение:

- `String` в цикле создает новый объект на каждой итерации (O(n²) по памяти/времени).
- `StringBuilder` использует один массив, расширяя его при необходимости (O(n)).
- **Вывод**: StringBuilder в разы быстрее (например, 1000 чисел: String ~1ms, StringBuilder ~0.01ms).

Пример 2: Многопоточное логирование (StringBuffer)

Сценарий: Несколько потоков записывают лог в общий буфер.

```
import java.util.concurrent.*;

public class StringBufferLoggingDemo {
    public static void main(String[] args) throws InterruptedException {
        StringBuffer logBuffer = new StringBuffer();
        ExecutorService executor = Executors.newFixedThreadPool(3);

        for (int i = 0; i < 3; i++) {
            final int threadId = i;
            executor.submit(() -> {
```

```
                for (int j = 0; j < 5; j++) {
                    logBuffer.append("Thread-").append(threadId).append(": Log
").append(j).append("\n");
                }
            });
        }

        executor.shutdown();
        executor.awaitTermination(5, TimeUnit.SECONDS);

        System.out.println("Log output:\n" + logBuffer.toString());
    }
}
```

Объяснение:

- `StringBuffer` безопасно используется в многопоточной среде (методы synchronized).
- Каждый поток добавляет записи в общий буфер.
- **Вывод**: Лог упорядочен, без data race (например, "Thread-0: Log 0\nThread-1: Log 0\n...").
- **Замечание**: Если использовать StringBuilder здесь, возможны повреждения данных (перемешивание строк).

Пример 3: Реверс строки и форматирование

Сценарий: Реверс строки и создание сложного формата (например, JSON-подобного).

```
public class StringBuilderFormattingDemo {
    public static void main(String[] args) {
        String input = "Hello, World!";

        // Реверс строки
        StringBuilder sb = new StringBuilder(input);
        String reversed = sb.reverse().toString();
        System.out.println("Reversed: " + reversed); // !dlroW ,olleH

        // Форматирование JSON
        sb = new StringBuilder();
        sb.append("{\n")
          .append("  \"name\": \"Alice\",\n")
          .append("  \"age\": ").append(30).append(",\n")
          .append("  \"city\": \"New York\"\n")
          .append("}");
        System.out.println("JSON:\n" + sb.toString());
```

```
    }
}
```

Объяснение:

- `reverse()`: Удобный метод для реверса (O(n)).
- Fluent interface (`append().append()`) упрощает построение сложных строк.
- **Вывод**:

```
Reversed: !dlroW ,olleH
JSON:
{
  "name": "Alice",
  "age": 30,
  "city": "New York"
}
```

Пример 4: Оптимизация емкости

Сценарий: Работа с большими данными, минимизация перевыделений памяти.

```
public class StringBuilderCapacityDemo {
    public static void main(String[] args) {
        // Без оптимизации
        StringBuilder sb1 = new StringBuilder(); // capacity=16
        for (int i = 0; i < 1000; i++) {
            sb1.append("data");
        }
        System.out.println("Default capacity: " + sb1.capacity()); // >1000 (расширялось)

        // С оптимизацией
        StringBuilder sb2 = new StringBuilder(4000); // Предполагаем ~4 символа * 1000
        for (int i = 0; i < 1000; i++) {
            sb2.append("data");
        }
        System.out.println("Optimized capacity: " + sb2.capacity()); // 4000
        System.out.println("Length: " + sb2.length()); // 4000
    }
}
```

Объяснение:

- `capacity()`: Показывает размер внутреннего массива.

- Без указания емкости StringBuilder расширяет массив (capacity * 2 + 2), что требует копирования.
- Указав начальную емкость, минимизируем перевыделения (O(n) вместо O(n log n)).
- **Трюк**: Используйте `ensureCapacity(n)` для динамической настройки.

---

4. Подводные камни и лучшие практики

Подводные камни

1. **StringBuilder в многопоточной среде**: Без синхронизации (synchronized block) может привести к data race.

```
StringBuilder sb = new StringBuilder();
// Нельзя использовать в потоках без synchronized
synchronized (sb) {
    sb.append("data");
}
```

2. **Переиспользование**: Не очищайте StringBuilder через `new StringBuilder()`. Используйте `setLength(0)` для сброса.

```
StringBuilder sb = new StringBuilder("data");
sb.setLength(0); // Теперь пустой
```

3. **Емкость**: Слишком большая начальная емкость тратит память, слишком маленькая — время на расширение.
4. **StringBuffer overhead**: Не используйте в однопоточных приложениях (лишняя синхронизация).

Лучшие практики

1. **Выбирайте StringBuilder по умолчанию**: Используйте StringBuffer только при явной необходимости (многопоточность).
2. **Задавайте емкость**: Если размер известен, используйте `new StringBuilder(capacity)` или `ensureCapacity()`.
3. **Fluent interface**: Цепочка методов (`append().append()`) улучшает читаемость.
4. **Конвертация в String**: Вызывайте `toString()` только в конце.
5. **Проверка null**: Перед `append` проверяйте null для объектов.

```
String value = null;
sb.append(Objects.toString(value, "default")); // Безопасно
```

6. **Тестирование производительности**: Используйте System.nanoTime() для сравнения String vs StringBuilder.

---

5. Практическое сравнение производительности

Тест: Конкатенация 10,000 строк.

```
public class PerformanceDemo {
    public static void main(String[] args) {
        // String
        long startTime = System.nanoTime();
        String s = "";
        for (int i = 0; i < 10000; i++) {
            s += "x";
        }
        long stringTime = System.nanoTime() - startTime;

        // StringBuilder
        startTime = System.nanoTime();
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < 10000; i++) {
            sb.append("x");
        }
        String builderResult = sb.toString();
        long builderTime = System.nanoTime() - startTime;

        // StringBuffer
        startTime = System.nanoTime();
        StringBuffer sbf = new StringBuffer();
        for (int i = 0; i < 10000; i++) {
            sbf.append("x");
        }
        String bufferResult = sbf.toString();
        long bufferTime = System.nanoTime() - startTime;

        System.out.println("String time: " + stringTime + " ns");
        System.out.println("StringBuilder time: " + builderTime + " ns");
        System.out.println("StringBuffer time: " + bufferTime + " ns");
    }
}
```

Примерный вывод (зависит от системы):

```
String time: 123456789 ns (~123ms)
StringBuilder time: 456789 ns (~0.45ms)
StringBuffer time: 567890 ns (~0.56ms)
```

Вывод: StringBuilder быстрее String в ~200 раз, StringBuffer чуть медленнее из-за синхронизации.

---

6. Пример из реального мира

Сценарий: Парсинг лога и создание отчета в формате HTML.

```
import java.util.*;

public class LogParserDemo {
    public static String parseLogToHtml(List<String> logEntries) {
        StringBuilder html = new StringBuilder(1000); // Предполагаем размер
        html.append("<html><body><h1>Log Report</h1><table border='1'>")
            .append("<tr><th>Index</th><th>Entry</th></tr>");

        for (int i = 0; i < logEntries.size(); i++) {
            html.append("<tr><td>").append(i + 1).append("</td><td>")
                .append(logEntries.get(i).replace("<", "&lt;")) // Экранирование
                .append("</td></tr>");
        }

        html.append("</table></body></html>");
        return html.toString();
    }

    public static void main(String[] args) {
        List<String> logs = Arrays.asList(
            "INFO: Server started",
            "ERROR: Connection failed",
            "DEBUG: Processing request"
        );
        String htmlReport = parseLogToHtml(logs);
        System.out.println(htmlReport);
    }
}
```

Объяснение:

- StringBuilder для построения HTML (эффективно, одна строка в конце).
- Экранирование `<` для корректного HTML.
- Начальная емкость 1000 для минимизации расширений.
- **Вывод**:

```
<html><body><h1>Log Report</h1><table border='1'><tr><th>Index</th>
<th>Entry</th></tr><tr><td>1</td><td>INFO: Server started</td></tr><tr>
<td>2</td><td>ERROR: Connection failed</td></tr><tr><td>3</td><td>DEBUG:
Processing request</td></tr></table></body></html>
```

---

7. Подводные камни в реальных проектах

- **Многократное toString()**: Вызывайте один раз в конце, иначе теряется производительность.
- **StringBuilder в пуле объектов**: Не используйте в пуле (например, Apache Commons Pool), так как состояние mutable.
- **Переполнение емкости**: Для очень больших строк (гигабайты) проверяйте `length()` против Integer.MAX_VALUE.
- **Null в append**: StringBuilder/StringBuffer обрабатывают null как "null". Проверяйте, если нужен другой default.

---

8. Ссылки для углубления

- Oracle Docs (StringBuilder): https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/StringBuilder.html
- Oracle Docs (StringBuffer): https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/StringBuffer.html
- Baeldung (Performance): https://www.baeldung.com/java-string-concatenation
- GeeksforGeeks (StringBuilder vs StringBuffer): https://www.geeksforgeeks.org/stringbuilder-class-in-java-with-examples/
- Medium (String Optimization): https://medium.com/@aarkay0106/java-string-performance-optimizations-3b8c4f8c7a1f

---

9. Заключение

- **StringBuilder**: Используйте для однопоточных приложений, где важна скорость (логи, парсинг, JSON).
- **StringBuffer**: Для многопоточных сценариев, но редко (альтернативы: `synchronized StringBuilder`, `ConcurrentStringBuilder` в сторонних библиотеках).
- **Трюки**: Задавайте емкость, используйте fluent interface, избегайте String конкатенации в циклах.

Если нужны дополнительные примеры (например, интеграция с regex, работа с большими файлами) или разбор конкретных задач — напишите!

Коллекции (особое внимание)

Коллекции — фреймворк для хранения данных (java.util). Основаны на интерфейсах: Collection, List, Set, Map. Поддерживают generics для type-safety (полиморфизм).

Фундаментальная разница ArrayList и LinkedList (оба реализуют List):

- **ArrayList**: На основе массива. Быстрый доступ по индексу (O(1)), медленные вставки/удаления в середину (O(n), сдвиг). Подходит для чтения, поиска.
- **LinkedList**: На основе связного списка (узлы с ссылками). Быстрые вставки/удаления (O(1)), медленный доступ по индексу (O(n), traversal). Подходит для очередей, стэков.

Пример:

```
import java.util.ArrayList;
import java.util.LinkedList;

ArrayList<String> al = new ArrayList<>();  // Создание
al.add("item");                            // Добавление
al.get(0);                                 // Доступ

LinkedList<String> ll = new LinkedList<>();
ll.addFirst("first");                      // Специфично для LinkedList
```

Другие: HashSet (уникальные, order), TreeSet (sorted), HashMap (key-value).

Stream API (особое внимание)

Stream API (с Java 8) — для функциональной обработки коллекций (функциональная парадигма в ООП). Streams — последовательности элементов, поддерживают lazy-операции, parallel.

Как создавать: collection.stream() или Stream.of(elements).

Примеры:

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.Arrays;

List<String> list = Arrays.asList("apple", "banana", "cherry");

// Фильтр и map
List<String> filtered = list.stream()
    .filter(s -> s.startsWith("a"))  // Промежуточная (lazy)
    .map(String::toUpperCase)        // Промежуточная
    .collect(Collectors.toList());   // Терминальная
```

```
// Reduce
int sum = Arrays.asList(1,2,3).stream().reduce(0, Integer::sum);

// Parallel
list.parallelStream().forEach(System.out::println);  // Многопоточно
```

Методы (основные):

- filter(Predicate predicate) — Фильтрует по условию.
- map(Function<T, R> mapper) — Преобразует элементы.
- flatMap(Function<T, Stream> mapper) — Разворачивает стримы.
- distinct() — Удаляет дубликаты.
- sorted() — Сортирует.
- limit(long maxSize) — Ограничивает размер.
- skip(long n) — Пропускает n элементов.
- forEach(Consumer action) — Для каждого.
- collect(Collector<T, A, R> collector) — Собирает в коллекцию.
- reduce(T identity, BinaryOperator accumulator) — Редукция.
- anyMatch(Predicate predicate) — Есть ли совпадение.
- allMatch(Predicate predicate) — Все совпадают.
- noneMatch(Predicate predicate) — Ни один не совпадает.
- findFirst() — Первый элемент.
- count() — Количество.

Streams не меняют исходную коллекцию, immutable по природе. Для parallel — используйте с caution (thread-safety). Ссылки: Oracle Tutorial — https://docs.oracle.com/javase/tutorial/collections/streams/index.html; Baeldung — https://www.baeldung.com/java-8-streams.

Модуль 2

Алгоритмы

Алгоритмы — это последовательность шагов для решения задачи, фундаментальная часть программирования в Java (и других языках). В Java они часто реализуются в методах или классах, используя структуры данных из коллекций. Алгоритмы классифицируются по типам: сортировка, поиск, графы, динамическое программирование и т.д. Они оцениваются по эффективности (время, пространство) с помощью Big O notation. В Java стандартные алгоритмы встроены (например, Arrays.sort()), но понимание их реализации важно для оптимизации. Принципы ООП применяются: абстракция в интерфейсах как Comparator для сортировки.

Подробнее: GeeksforGeeks — https://www.geeksforgeeks.org/fundamentals-of-algorithms/ (актуально на 2025).

Big O notation (какие виды временной сложности алгоритмов бывают)

Big O notation описывает асимптотическую сложность алгоритма — как растет время/память с ростом входных данных (n). Это верхняя граница worst-case. Виды временной сложности:

- **O(1) — Константа**: Время не зависит от n. Пример: доступ к элементу массива по индексу.
- **O(log n) — Логарифмическая**: Время растет логарифмически (половинение данных). Пример: бинарный поиск в отсортированном массиве.
- **O(n) — Линейная**: Пропорционально n. Пример: линейный поиск.
- **O(n log n) — Линейно-логарифмическая**: Типична для эффективных сортировок. Пример: merge sort.
- **O(n²) — Квадратичная**: Для вложенных циклов. Пример: bubble sort.
- **O(2^n) — Экспоненциальная**: Быстро растет, для brute-force. Пример: рекурсивный Fibonacci без мемоизации.
- **O(n!) — Факториальная**: Очень медленно, для перестановок.

Также есть Big Θ (средний) и Big Ω (лучший). В Java учитывайте для коллекций и алгоритмов. Пример: Arrays.sort() — O(n log n). Ссылки: Baeldung — https://www.baeldung.com/java-algorithm-complexity.

Виды сортировок (отличие quicksort от merge sort)

Сортировки — алгоритмы упорядочивания элементов. В Java — Arrays.sort() или Collections.sort(). Виды:

- **Bubble Sort**: Сравнивает соседние, swaps. O(n²), простая.
- **Insertion Sort**: Вставляет в отсортированную часть. O(n²), хороша для малых массивов.
- **Selection Sort**: Выбирает минимум, swaps. O(n²).
- **Merge Sort**: Divide-and-conquer: делит, сортирует, merges. O(n log n), стабильная, требует доп. памяти O(n).
- **Quick Sort**: Выбирает pivot, партиционирует, рекурсия. Средний O(n log n), worst O(n²), in-place, нестабильная.
- **Heap Sort**: Строит heap, извлекает max. O(n log n), in-place.
- **Tim Sort**: Гибрид merge+insertion, используется в Java's Arrays.sort(). O(n log n).

Отличие QuickSort от MergeSort: QuickSort быстрее в среднем, in-place (меньше памяти), но worst-case O(n²) (если плохой pivot); MergeSort всегда O(n log n), стабильный (сохраняет порядок

равных), но требует O(n) памяти. QuickSort — для скорости, MergeSort — для стабильности. Пример кода MergeSort:

```
public class MergeSort {
    void merge(int[] arr, int l, int m, int r) {
        int n1 = m - l + 1;
        int n2 = r - m;
        int[] L = new int[n1];
        int[] R = new int[n2];
        for (int i = 0; i < n1; ++i) L[i] = arr[l + i];
        for (int j = 0; j < n2; ++j) R[j] = arr[m + 1 + j];
        int i = 0, j = 0, k = l;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) arr[k++] = L[i++];
            else arr[k++] = R[j++];
        }
        while (i < n1) arr[k++] = L[i++];
        while (j < n2) arr[k++] = R[j++];
    }

    void sort(int[] arr, int l, int r) {
        if (l < r) {
            int m = l + (r - l) / 2;
            sort(arr, l, m);
            sort(arr, m + 1, r);
            merge(arr, l, m, r);
        }
    }
}
```

Для QuickSort:

```
public class QuickSort {
    int partition(int[] arr, int low, int high) {
        int pivot = arr[high];
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
```

```
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;
        return i + 1;
    }

    void sort(int[] arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high);
            sort(arr, low, pi - 1);
            sort(arr, pi + 1, high);
        }
    }
}
```

Ссылки: Oracle Docs на Arrays — https://docs.oracle.com/javase/21/docs/api/java/util/Arrays.html#sort(int[]).

Реализация часто используемых алгоритмов с объяснением и Big O

В программировании, особенно в Java, некоторые алгоритмы используются чаще других из-за их универсальности и эффективности. Я выбрал **шесть наиболее популярных алгоритмов**, которые регулярно встречаются в реальной разработке: **сортировка (QuickSort)**, **поиск (Binary Search)**, **DFS (поиск в глубину)**, **BFS (поиск в ширину)**, **Two Pointers** и **Sliding Window**. Для каждого я предоставлю:

- **Описание и назначение**: Что делает алгоритм и где применяется.
- **Реализация на Java**: Полный, рабочий код с комментариями.
- **Пошаговое объяснение**: Что происходит в коде.
- **Асимптотическая сложность (Big O)**: Время и память.
- **Сценарий использования**: Реальный пример из разработки.
- **Ссылки**: Достоверные источники (проверены на актуальность на октябрь 2025).

Я выбрал эти алгоритмы, так как они охватывают разные категории (сортировка, поиск, графы, оптимизация) и часто встречаются в задачах LeetCode, интервью и реальных проектах. Код будет компактным, но подробным, с акцентом на ясность.

---

1. QuickSort (Быстрая сортировка)

Описание и назначение: QuickSort — алгоритм сортировки, использующий принцип "разделяй и властвуй". Выбирает опорный элемент (pivot), разделяет массив на части (меньше и больше

pivot), рекурсивно сортирует их. Используется в коллекциях (Java's `Arrays.sort`), базах данных, UI-отрисовке.

Сценарий использования: Сортировка списка пользователей по ID, оптимизация запросов в БД.

Код:

```
public class QuickSort {
    public void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high); // Разделение
            quickSort(arr, low, pi - 1); // Сортировка левой части
            quickSort(arr, pi + 1, high); // Сортировка правой части
        }
    }

    private int partition(int[] arr, int low, int high) {
        int pivot = arr[high]; // Опорный элемент
        int i = low - 1; // Индекс меньшего элемента
        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) {
                i++;
                // Обмен
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        // Помещаем pivot на место
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;
        return i + 1;
    }

    public static void main(String[] args) {
        int[] arr = {64, 34, 25, 12, 22, 11, 90};
        QuickSort qs = new QuickSort();
        qs.quickSort(arr, 0, arr.length - 1);
        System.out.println(Arrays.toString(arr)); // [11, 12, 22, 25, 34, 64, 90]
    }
}
```

Пошаговое объяснение:

1. **Выбор pivot**: Берем последний элемент (`arr[high]`).
2. **Partition**: Перемещаем элементы < pivot влево, > pivot вправо. `i` указывает на границу меньших элементов.
3. **Обмен**: Если `arr[j] <= pivot`, меняем с `arr[i+1]`, увеличиваем `i`.
4. **Помещение pivot**: После цикла ставим pivot на место (`i+1`).
5. **Рекурсия**: Сортируем подмассивы до и после pivot.

Big O:

- **Время**:
  - Лучший случай: O(n log n) — массив делится пополам.
  - Средний случай: O(n log n).
  - Худший случай: O(n²) — если pivot всегда минимальный/максимальный (например, уже отсортированный массив).
- **Память**: O(log n) — для стека рекурсии (в среднем). Может быть O(n) в худшем случае.

Трюки:

- Выбор pivot (медиана, random) снижает вероятность O(n²).
- Для малых массивов (<50) лучше Insertion Sort.

Ссылки:

- GeeksforGeeks: https://www.geeksforgeeks.org/quick-sort/
- Baeldung: https://www.baeldung.com/java-quicksort

---

2. Binary Search (Бинарный поиск)

Описание и назначение: Ищет элемент в отсортированном массиве, деля его пополам на каждой итерации. Используется в индексах БД, поиске по ID, автодополнении.

Сценарий использования: Поиск пользователя по ID в отсортированном списке.

Код:

```
public class BinarySearch {
    public int binarySearch(int[] arr, int target) {
        int left = 0, right = arr.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2; // Избегаем переполнения
            if (arr[mid] == target) {
                return mid;
            } else if (arr[mid] < target) {
```

```
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return -1; // Не найдено
    }

    public static void main(String[] args) {
        int[] arr = {2, 3, 4, 10, 40};
        BinarySearch bs = new BinarySearch();
        int result = bs.binarySearch(arr, 10);
        System.out.println("Index of 10: " + result); // 3
    }
}
```

Пошаговое объяснение:

1. **Инициализация**: `left` — начало, `right` — конец.
2. **Средняя точка**: `mid = left + (right - left) / 2` (безопасно от переполнения).
3. **Сравнение**: Если `arr[mid] == target`, возвращаем `mid`. Если меньше — ищем справа (`left = mid + 1`), если больше — слева (`right = mid - 1`).
4. **Повторение**: Пока `left <= right`.

Big O:

- **Время**: O(log n) — деление массива пополам.
- **Память**: O(1) для итеративной версии, O(log n) для рекурсивной (стек вызовов).

Трюки:

- Требуется отсортированный массив.
- Для больших данных рекурсивная версия менее эффективна.

Ссылки:

- GeeksforGeeks: https://www.geeksforgeeks.org/binary-search/
- Baeldung: https://www.baeldung.com/java-binary-search

---

3. Depth-First Search (DFS, Поиск в глубину)

Описание и назначение: Обходит граф/дерево, углубляясь в каждую ветку до конца перед возвратом. Используется для топологической сортировки, поиска циклов, путей в графе (например, в роутинге).

Сценарий использования: Проверка связности компонент в графе друзей в соцсети.

Код (рекурсивный DFS для графа):

```
import java.util.*;

public class DFS {
    private Map<Integer, List<Integer>> graph = new HashMap<>();

    public void addEdge(int u, int v) {
        graph.computeIfAbsent(u, k -> new ArrayList<>()).add(v);
        graph.computeIfAbsent(v, k -> new ArrayList<>()).add(u); // Ненаправленный
граф
    }

    public void dfs(int start, Set<Integer> visited) {
        visited.add(start);
        System.out.print(start + " ");
        for (int neighbor : graph.getOrDefault(start, Collections.emptyList())) {
            if (!visited.contains(neighbor)) {
                dfs(neighbor, visited);
            }
        }
    }

    public static void main(String[] args) {
        DFS dfs = new DFS();
        dfs.addEdge(0, 1);
        dfs.addEdge(0, 2);
        dfs.addEdge(1, 3);
        dfs.addEdge(2, 4);
        dfs.dfs(0, new HashSet<>()); // 0 1 3 2 4
    }
}
```

Пошаговое объяснение:

1. **Граф**: Хранится как список смежности (`Map<Integer, List<Integer>>`).
2. **Посещение**: Помечаем вершину как visited, выводим.
3. **Рекурсия**: Для каждого непосещенного соседа вызываем DFS.
4. **База**: Если вершина посещена, пропускаем.

Big O:

- **Время**: O(V + E) — V вершин, E рёбер (каждая вершина/ребро обрабатывается раз).
- **Память**: O(V) для visited + O(V) для стека рекурсии.

Трюки:

- Итеративная версия (со стеком) экономит память при глубоких графах.
- Для циклов добавьте проверку parent.

Ссылки:

- GeeksforGeeks: https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/
- Baeldung: https://www.baeldung.com/java-depth-first-search

---

4. Breadth-First Search (BFS, Поиск в ширину)

Описание и назначение: Обходит граф/дерево по уровням, исследуя соседей перед углублением. Используется для кратчайших путей (в невзвешенных графах), анализа сетей.

Сценарий использования: Поиск ближайших друзей в соцсети.

Код:

```
import java.util.*;

public class BFS {
    private Map<Integer, List<Integer>> graph = new HashMap<>();

    public void addEdge(int u, int v) {
        graph.computeIfAbsent(u, k -> new ArrayList<>()).add(v);
        graph.computeIfAbsent(v, k -> new ArrayList<>()).add(u);
    }

    public void bfs(int start) {
        Set<Integer> visited = new HashSet<>();
        Queue<Integer> queue = new LinkedList<>();
        visited.add(start);
        queue.offer(start);

        while (!queue.isEmpty()) {
            int vertex = queue.poll();
            System.out.print(vertex + " ");
            for (int neighbor : graph.getOrDefault(vertex,
Collections.emptyList())) {
                if (!visited.contains(neighbor)) {
                    visited.add(neighbor);
```

```
                    queue.offer(neighbor);
                }
            }
        }
    }

    public static void main(String[] args) {
        BFS bfs = new BFS();
        bfs.addEdge(0, 1);
        bfs.addEdge(0, 2);
        bfs.addEdge(1, 3);
        bfs.addEdge(2, 4);
        bfs.bfs(0); // 0 1 2 3 4
    }
}
```

Пошаговое объяснение:

1. **Очередь**: Добавляем стартовую вершину.
2. **Обработка**: Извлекаем вершину, выводим, добавляем непосещенных соседей в очередь.
3. **Повторение**: Пока очередь не пуста.
4. **Посещение**: Отмечаем вершины, чтобы избежать циклов.

Big O:

- **Время**: O(V + E) — каждая вершина/ребро обрабатывается раз.
- **Память**: O(V) для visited и queue.

Трюки:

- Для кратчайшего пути храните distances и parents.
- Используйте Deque для оптимизации.

Ссылки:

- GeeksforGeeks: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
- Baeldung: https://www.baeldung.com/java-breadth-first-search

---

5. Two Pointers (Два указателя)

Описание и назначение: Использует два указателя для обхода массива/списка, решая задачи за O(n). Применяется для поиска пар, удаления дубликатов, проверки палиндромов.

Сценарий использования: Удаление дубликатов из отсортированного массива (LeetCode #26).

Код:

```
public class TwoPointers {
    public int removeDuplicates(int[] nums) {
        if (nums.length == 0) return 0;
        int write = 1; // Указатель для записи уникальных элементов
        for (int read = 1; read < nums.length; read++) {
            if (nums[read] != nums[read - 1]) {
                nums[write] = nums[read];
                write++;
            }
        }
        return write;
    }

    public static void main(String[] args) {
        int[] nums = {0, 0, 1, 1, 1, 2, 2, 3, 3, 4};
        TwoPointers tp = new TwoPointers();
        int len = tp.removeDuplicates(nums);
        System.out.println("Length: " + len); // 5
        System.out.println(Arrays.toString(Arrays.copyOf(nums, len))); // [0, 1, 2, 3, 4]
    }
}
```

Пошаговое объяснение:

1. **Инициализация**: `write` указывает на позицию для записи уникального элемента.
2. **Чтение**: `read` проверяет каждый элемент.
3. **Сравнение**: Если `nums[read] != nums[read-1]`, записываем в `nums[write]` и сдвигаем `write`.
4. **Результат**: `write` — длина уникального массива.

Big O:

- **Время**: O(n) — один проход по массиву.
- **Память**: O(1) — изменения in-place.

Трюки:

- Работает только с отсортированным массивом.
- Варианты: Fast/Slow pointers (для циклов), Left/Right (для палиндромов).

Ссылки:

- LeetCode: https://leetcode.com/problems/remove-duplicates-from-sorted-array/

- GeeksforGeeks: https://www.geeksforgeeks.org/two-pointers-technique/

---

6. Sliding Window (Скользящее окно)

Описание и назначение: Использует окно переменного/фиксированного размера для обработки подмассивов/подстрок. Применяется для поиска подстрок, максимума/минимума в окне.

Сценарий использования: Найти самую длинную подстроку без повторяющихся символов (LeetCode #3).

Код:

```
import java.util.*;

public class SlidingWindow {
    public int lengthOfLongestSubstring(String s) {
        Map<Character, Integer> map = new HashMap<>();
        int maxLen = 0, left = 0;
        for (int right = 0; right < s.length(); right++) {
            char c = s.charAt(right);
            if (map.containsKey(c) && map.get(c) >= left) {
                left = map.get(c) + 1; // Сдвигаем окно
            } else {
                maxLen = Math.max(maxLen, right - left + 1);
            }
            map.put(c, right);
        }
        return maxLen;
    }

    public static void main(String[] args) {
        SlidingWindow sw = new SlidingWindow();
        String s = "abcabcbb";
        System.out.println("Longest substring length: " +
sw.lengthOfLongestSubstring(s)); // 3 (abc)
    }
}
```

Пошаговое объяснение:

1. **Окно**: `left` и `right` определяют границы.
2. **HashMap**: Хранит индексы символов.
3. **Сдвиг**: Если символ повторяется внутри окна, сдвигаем `left` за последнее вхождение.
4. **Обновление**: Максимальная длина окна — `right - left + 1`.

5. **Хранение**: Обновляем индекс символа в map.

Big O:

- **Время**: O(n) — каждый символ добавляется/удаляется раз.
- **Память**: O(min(m, n)) — m размер алфавита, n длина строки.

Трюки:

- Для фиксированного окна используйте массив вместо HashMap.
- Оптимизируйте для ASCII (массив char[128]).

Ссылки:

- LeetCode: https://leetcode.com/problems/longest-substring-without-repeating-characters/
- Baeldung: https://www.baeldung.com/java-sliding-window

Общие замечания

- **Оптимизация**: Для реальных проектов профилируйте (например, VisualVM), чтобы выбрать подходящий алгоритм.
- **Тестирование**: Проверяйте edge cases (пустой массив, граф без рёбер, строка с повторами).
- **Модульность**: Инкапсулируйте алгоритмы в классы/методы для переиспользования.

Примеры в реальной разработке:

- **QuickSort**: Сортировка данных в отчетах (JasperReports).
- **Binary Search**: Поиск в индексах Elasticsearch.
- **DFS/BFS**: Анализ зависимостей в микросервисах (Spring Cloud).
- **Two Pointers**: Парсинг логов (удаление дубликатов).
- **Sliding Window**: Обработка потоков данных (Apache Kafka).

Если нужны другие алгоритмы (например, Dijkstra, Merge Sort) или разбор конкретной задачи (LeetCode, проект) — напишите!

Generics

Generics — параметризованные типы для type-safety и переиспользования (полиморфизм в ООП). Введены в Java 5. Позволяют классам/методам работать с разными типами без casting. Синтаксис: (type parameter).

Пример:

```
class Box<T> {  // Generic класс
    private T item;
```

```
    public void set(T item) { this.item = item; }
    public T get() { return item; }
}

Box<String> stringBox = new Box<>();  // Type inference
stringBox.set("hello");
String s = stringBox.get();  // Без cast
```

Wildcards: <? extends T> (upper bound, read-only), <? super T> (lower bound, write). Плюсы: Компилятор проверяет типы, избегает ClassCastException. Минусы: Сложность с primitives (используйте wrappers). Ссылки: Oracle Tutorial — https://docs.oracle.com/javase/tutorial/java/generics/index.html.

Коллекции

Коллекции — фреймворк (java.util) для хранения/манипуляции группами объектов. Поддерживают generics. Основаны на интерфейсах для абстракции. Используют хэширование, деревья для эффективности. Ранее упоминалось, так что добавлю: Коллекции mutable по умолчанию, но есть immutable. Принцип полиморфизма: List list = new ArrayList<>();. Ссылки: Baeldung — https://www.baeldung.com/java-collections.

Иерархия коллекций

Иерархия:

- **Collection** (интерфейс): Базовый для List, Set, Queue. Методы: add, remove, contains, size, iterator.
- **List**: Упорядоченная, дубликаты. Реализации: ArrayList, LinkedList, Vector.
- **Set**: Уникальные элементы, нет порядка (кроме SortedSet). Реализации: HashSet, LinkedHashSet, TreeSet (SortedSet).
- **Queue**: FIFO. Реализации: LinkedList, PriorityQueue.
- **Map** (не extends Collection): Key-value. Реализации: HashMap, LinkedHashMap, TreeMap (SortedMap).
- **Iterable**: Базовый для всех, для foreach.

Пример: Collection < String > col = new ArrayList<>(); — полиморфизм.

List

List — упорядоченная коллекция с дубликатами, доступ по индексу. Методы (основные):

- public boolean add(E e) — Добавляет элемент в конец.
- public void add(int index, E element) — Добавляет по индексу.

- public E get(int index) — Получает по индексу.
- public E set(int index, E element) — Заменяет по индексу.
- public boolean remove(Object o) — Удаляет первое вхождение.
- public E remove(int index) — Удаляет по индексу.
- public int size() — Размер.
- public boolean isEmpty() — Пустой ли.
- public void clear() — Очищает.
- public boolean contains(Object o) — Содержит ли.
- public int indexOf(Object o) — Индекс первого.
- public int lastIndexOf(Object o) — Индекс последнего.
- public Iterator< E > iterator() — Итератор.
- public ListIterator< E > listIterator() — ListIterator для bidirectional.

Реализации: ArrayList (массив), LinkedList (список). Ссылки: Oracle — https://docs.oracle.com/javase/21/docs/api/java/util/List.html.

Set

Set — уникальные элементы, нет дубликатов (equals/hashCode). Нет индекса. Методы (наследует от Collection):

- public boolean add(E e) — Добавляет, если нет.
- public boolean remove(Object o) — Удаляет.
- public boolean contains(Object o) — Содержит ли.
- public int size() — Размер.
- public boolean isEmpty() — Пустой.
- public void clear() — Очищает.
- public Iterator< E > iterator() — Итератор.

Реализации: HashSet (хэш, O(1) операции), LinkedHashSet (порядок вставки), TreeSet (sorted, O(log n)). Ссылки: Oracle — https://docs.oracle.com/javase/21/docs/api/java/util/Set.html.

Map

Map — пары key-value, ключи уникальные. Не Collection. Методы:

- public V put(K key, V value) — Добавляет/заменяет.
- public V get(Object key) — Получает value.
- public V remove(Object key) — Удаляет.

- public boolean containsKey(Object key) — Есть ключ.
- public boolean containsValue(Object value) — Есть value.
- public int size() — Размер.
- public boolean isEmpty() — Пустой.
- public void clear() — Очищает.
- public Set< K > keySet() — Ключи.
- public Collection< V > values() — Значения.
- public Set<Map.Entry<K,V>> entrySet() — Entries.

Реализации: HashMap (хэш), LinkedHashMap (порядок), TreeMap (sorted). Ссылки: Oracle — https://docs.oracle.com/javase/21/docs/api/java/util/Map.html.

Временная сложность для разных операций разных коллекций

Сложность (average case):

- **ArrayList**:
  - add (конец): O(1) amortized.
  - get/set: O(1).
  - remove (индекс): O(n).
  - contains: O(n).
- **LinkedList**:
  - add (конец/начало): O(1).
  - get/set: O(n).
  - remove (индекс): O(n).
  - contains: O(n).
- **HashSet/HashMap**:
  - add/put: O(1).
  - get/contains: O(1).
  - remove: O(1).
- **LinkedHashSet/LinkedHashMap**:
  - Аналогично Hash, но итерация O(n) в порядке вставки.
- **TreeSet/TreeMap**:
  - add/put: O(log n).
  - get/contains: O(log n).

- remove: O(log n).

Worst-case для hash: O(n) при коллизиях, но в Java 8+ — treeify для улучшения. Ссылки: GeeksforGeeks — https://www.geeksforgeeks.org/time-complexities-of-all-java-collections/.

Неизменяемые коллекции (создание с помощью List.of(), методы класса Collections)

Immutable коллекции — нельзя модифицировать после создания (для thread-safety, immutable objects). С Java 9: Фабричные методы как List.of(), Set.of(), Map.of().

Пример:

```
List<String> immutableList = List.of("a", "b", "c");  // Нельзя add/remove
Set<String> immutableSet = Set.of("a", "b");
Map<String, Integer> immutableMap = Map.of("key1", 1, "key2", 2);
```

Методы Collections:

- Collections.unmodifiableList(List<? extends T> list) — Делает view immutable.
- Аналогично для Set, Map, Collection.

Пример:

```
List<String> mutable = new ArrayList<>(Arrays.asList("a", "b"));
List<String> immutable = Collections.unmodifiableList(mutable);  // Throws UnsupportedOperationException на modify
```

Плюсы: Безопасность, простота. Ссылки: Oracle — https://docs.oracle.com/javase/21/docs/api/java/util/List.html#of(E...).

LinkedHashMap, TreeMap

- **LinkedHashMap**: Extends HashMap, сохраняет порядок вставки (или доступа с accessOrder=true). O(1) операции, итерация в порядке.

Пример:

```
LinkedHashMap<String, Integer> lhm = new LinkedHashMap<>();
lhm.put("one", 1);
lhm.put("two", 2);  // Порядок: one, two
```

- **TreeMap**: Implements SortedMap, ключи sorted (natural или Comparator). O(log n) операции.

Пример:

```
TreeMap<String, Integer> tm = new TreeMap<>();
tm.put("b", 2);
tm.put("a", 1);  // Sorted: a, b
```

Используйте LinkedHashMap для LRU-cache, TreeMap для sorted данных. Ссылки: Baeldung — https://www.baeldung.com/java-linkedhashmap-vs-treemap.

Интерфейсы Comparable и Comparator

Для сортировки (полиморфизм).

- **Comparable**: Natural ordering. Метод: int compareTo(T o). Реализуется в классе.

Пример:

```
class Person implements Comparable<Person> {
    String name;
    @Override
    public int compareTo(Person other) {
        return this.name.compareTo(other.name);
    }
}
```

- **Comparator**: External ordering. Метод: int compare(T o1, T o2). Для Collections.sort(list, comparator).

Пример:

```
Comparator<Person> byName = (p1, p2) -> p1.name.compareTo(p2.name);
```

Отличие: Comparable — внутри класса, один ordering; Comparator — внешний, множественные. Ссылки: Oracle — https://docs.oracle.com/javase/tutorial/collections/interfaces/order.html.

Лямбда выражения, функциональные интерфейсы, Stream API

Ранее упоминались, так что более развернутый пример.

Лямбда — анонимные функции (функциональная парадигма). Синтаксис: (params) -> body.

Функциональный интерфейс — с одним абстрактным методом (SAM), как Runnable, Comparator.

Пример лямбды:

```
Runnable r = () -> System.out.println("Hello");  // Лямбда для Runnable
new Thread(r).start();
```

Stream API: Развернутый пример обработки списка людей.

```
import java.util.*;
import java.util.stream.*;

class Person {
    String name;
    int age;
    Person(String name, int age) { this.name = name; this.age = age; }
}

public class StreamExample {
    public static void main(String[] args) {
        List<Person> people = Arrays.asList(
            new Person("Alice", 25),
            new Person("Bob", 30),
            new Person("Charlie", 20),
            new Person("David", 35)
        );

        // Фильтр (возраст > 25), map (имена uppercase), сортировка, сбор в Map (имя:возраст)
        Map<String, Integer> result = people.stream()
            .filter(p -> p.age > 25)                        // Промежуточная: фильтр
            .sorted(Comparator.comparingInt(p -> p.age))    // Промежуточная: сортировка по возрасту
            .map(p -> {                                     // Промежуточная: map с лямбдой
                p.name = p.name.toUpperCase();
                return p;
            })
            .collect(Collectors.toMap(                      // Терминальная: в Map
                p -> p.name,                                // Key mapper
                p -> p.age,                                 // Value mapper
                (old, newVal) -> old                        // Merge function для дубликатов
            ));

        System.out.println(result);  // {BOB=30, DAVID=35}

        // Parallel stream с reduce: сумма возрастов
        int totalAge = people.parallelStream()
            .mapToInt(p -> p.age)                           // Map в int
```

```
            .reduce(0, Integer::sum);                    // Reduce с method
reference

        System.out.println("Total age: " + totalAge);

        // Grouping by: Группировка по возрасту >30
        Map<Boolean, List<Person>> grouped = people.stream()
            .collect(Collectors.partitioningBy(p -> p.age > 30));

        System.out.println(grouped);
    }
}
```

Это показывает chaining, parallel, collectors. Ссылки: Baeldung — https://www.baeldung.com/java-8-lambda-expressions-tips; Oracle Stream — https://docs.oracle.com/javase/21/docs/api/java/util/stream/package-summary.html.

Модуль 3

Исключения

Исключения в Java — это события, которые прерывают нормальный поток выполнения программы, сигнализируя об ошибке или необычном состоянии. Они являются объектами, наследующими от класса Throwable, и поддерживают принцип абстракции в ООП, позволяя обрабатывать ошибки централизованно. Исключения делятся на checked (компилятор требует обработки) и unchecked (runtime-ошибки). Пример: ArithmeticException при делении на ноль. Использование исключений улучшает читаемость кода, отделяя логику от обработки ошибок. Подробнее: Oracle Docs — https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html.

Иерархия исключений

Иерархия начинается с Throwable (implements Serializable):

- **Throwable**: Базовый класс для ошибок и исключений.
  - **Error**: Необрабатываемые ошибки (системные, как OutOfMemoryError). Не ловить.
  - **Exception**: Обрабатываемые.
    - **RuntimeException** (unchecked): Ошибки программиста, как NullPointerException, IndexOutOfBoundsException, ArithmeticException.
    - **Checked Exceptions**: Требуют try-catch или throws, как IOException, SQLException.

Пример:

```
Throwable
├── Error
│   ├── OutOfMemoryError
│   └── StackOverflowError
└── Exception
    ├── RuntimeException
    │   ├── NullPointerException
    │   ├── ArithmeticException
    │   └── IllegalArgumentException
    └── IOException
        ├── FileNotFoundException
        └── EOFException
```

Это дерево обеспечивает полиморфизм: catch(Exception e) ловит все подтипы.

Способы обработки исключений

Обработка — для graceful recovery или логирования.

- **try-catch**: Ловит исключение в блоке.
- **try-catch-finally**: Finally всегда выполняется (cleanup).
- **throws**: Пропагандирует вверх по стеку.
- **throw**: Бросает вручную.

Пример:

```
try {
    int x = 1 / 0;  // ArithmeticException
} catch (ArithmeticException e) {
    System.out.println("Error: " + e.getMessage());
} finally {
    System.out.println("Cleanup");
}

void method() throws IOException {  // throws
    throw new IOException("IO error");  // throw
}
```

Множественные catch: catch (Exception1 | Exception2 e) {}. Ссылки: Baeldung — https://www.baeldung.com/java-exceptions.

try с ресурсами

Try-with-resources (Java 7+) — для автоматического закрытия ресурсов (implements AutoCloseable). Упрощает код, предотвращает утечки.

Пример:

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
    String line = br.readLine();
} catch (IOException e) {
    // Обработка
}  // br.close() автоматически
```

Множественные: try (Res1 r1 = ...; Res2 r2 = ...) {}. Finally не нужен для close. Ссылки: Oracle — https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html.

Ввод - вывод

Ввод-вывод (I/O) — для работы с файлами, сетью, консолью. Java предоставляет потоки (streams) для байт/символов. Это абстрагирует устройства как последовательности данных. I/O может быть блокирующим, использовать буферы для эффективности. Принцип инкапсуляции: Классы скрывают детали. Подробнее: Oracle Tutorial — https://docs.oracle.com/javase/tutorial/essential/io/index.html.

Какие основные классы используются для работы с потоками ввода - вывода

- **Байтовые потоки** (для binary данных):
  - InputStream: Базовый для чтения байт (abstract).
  - OutputStream: Базовый для записи байт (abstract).
  - FileInputStream: Чтение из файла.
  - FileOutputStream: Запись в файл.
  - BufferedInputStream: Буферизация для эффективности.
  - BufferedOutputStream: Аналогично.
- **Символьные потоки** (для текста, Unicode):
  - Reader: Базовый для чтения символов.
  - Writer: Базовый для записи символов.
  - FileReader: Чтение текста из файла.
  - FileWriter: Запись текста в файл.
  - BufferedReader: Буферизованное чтение (readLine()).
  - BufferedWriter: Буферизованная запись.

    - PrintWriter: Удобная запись (print, println).
- **Другие**: DataInputStream/DataOutputStream (примитивы), ObjectInputStream/ObjectOutputStream (сериализация).

Пример:

```
try (BufferedReader br = new BufferedReader(new FileReader("input.txt"));
     PrintWriter pw = new PrintWriter(new FileWriter("output.txt"))) {
    String line;
    while ((line = br.readLine()) != null) {
        pw.println(line.toUpperCase());
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Основные правила во время работы с потоками ввода - вывода.

- **Закрывать потоки**: Всегда close() для освобождения ресурсов (try-with-resources).
- **Обработка исключений**: I/O throws IOException — ловить или throws.
- **Буферизация**: Использовать Buffered* для производительности (малые чтения дорогие).
- **Flush**: Вызывать flush() для немедленной записи (в BufferedWriter).
- **Кодировка**: Указывать charset (Charset.forName("UTF-8")) для текста.
- **Не смешивать байты/символы**: Выбрать тип потока.
- **Thread-safety**: Потоки не thread-safe — синхронизировать.
- **Проверка существования**: File.exists() перед открытием.

Нарушение приводит к утечкам, поврежденным файлам. Ссылки: Baeldung — https://www.baeldung.com/java-io-best-practices.

Сериализация

Сериализация — преобразование объекта в байтовый поток для хранения/передачи (файл, сеть). Десериализация — обратное. Класс должен implements Serializable (marker interface). Поля transient — не сериализуются, static — не (принадлежат классу).

Пример:

```
import java.io.*;

class Person implements Serializable {
```

```
    private static final long serialVersionUID = 1L;  // Для версионности
    String name;
    transient int password;  // Не сериализуется
}

public class SerializeDemo {
    public static void main(String[] args) {
        Person p = new Person();
        p.name = "Alice";

        try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("person.ser"))) {
            oos.writeObject(p);  // Сериализация
        } catch (IOException e) {
            e.printStackTrace();
        }

        try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("person.ser"))) {
            Person deserialized = (Person) ois.readObject();  // Десериализация
            System.out.println(deserialized.name);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Плюсы: Легко сохранять состояние. Минусы: Безопасность (readObject может быть уязвим). Ссылки: Oracle — https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html.

Сборщики проектов (Gradle, Maven)

Сборщики — инструменты для автоматизации сборки, зависимостей, тестов. Они управляют lifecycle проекта.

- **Maven**: XML-based (pom.xml). Стандартизированные фазы, центральный репозиторий (Maven Central). Плюсы: Конвенции, плагины. Минусы: Жесткий.
- **Gradle**: Groovy/Kotlin DSL (build.gradle). Гибкий, incremental builds. Плюсы: Производительность, кастомизация. Минусы: Кручевая кривая.

Пример Maven pom.xml:

```
<project>
    <groupId>com.example</groupId>
```

```
    <artifactId>my-app</artifactId>
    <version>1.0</version>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
        </dependency>
    </dependencies>
</project>
```

Gradle build.gradle:

```
plugins {
    id 'java'
}

dependencies {
    testImplementation 'junit:junit:4.13.2'
}
```

Ссылки: Maven — https://maven.apache.org/guides/; Gradle — https://docs.gradle.org/current/userguide/userguide.html.

Этапы сборки в Maven

Lifecycle Maven: clean, validate, compile, test, package, verify, install, deploy. Выполняются последовательно.

- **validate**: Проверяет проект.
- **compile**: Компилирует source.
- **test**: Запускает тесты.
- **package**: Пакует в JAR/WAR.
- **verify**: Проверки качества.
- **install**: Устанавливает в local repo.
- **deploy**: Загружает в remote repo.
- **clean**: Удаляет build-артефакты.

Команда: mvn clean install. Ссылки: Maven Docs — https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html.

Где на локальной машине хранятся скачанные зависимости

В local repository: По умолчанию ~/.m2/repository (Unix/Mac) или C:\Users\username.m2\repository (Windows). Зависимости хранятся как groupId/artifactId/version/artifactId-version.jar.

Пример: junit/junit/4.13.2/junit-4.13.2.jar.

Можно изменить в settings.xml. Gradle: ~/.gradle/caches. Ссылки: Maven — https://maven.apache.org/guides/mini/guide-configuring-maven.html.

Git

Git — распределенная система контроля версий для трекинга изменений в коде. Позволяет ветвление, слияние, collaboration. Не часть Java, но essential для разработчиков. Команды в терминале. Подробнее: Git Docs — https://git-scm.com/docs.

push, pull, fetch

- **push**: Отправляет локальные коммиты в remote repo. git push origin main.
- **pull**: Fetch + merge из remote. git pull origin main.
- **fetch**: Скачивает изменения из remote без merge. git fetch origin.

Пример:

```
git add .
git commit -m "Update"
git push origin main  # Push

git fetch origin      # Fetch
git merge origin/main # Manual merge after fetch

git pull origin main  # Fetch + merge
```

2 варианта слияния веток (merge, rebase)

- **merge**: Создает merge-коммит, сохраняет историю. git merge feature.
- **rebase**: Перемещает коммиты на top master, linear история. git rebase main.

Плюсы merge: Сохраняет контекст. Минусы: Messy история.

Плюсы rebase: Clean история. Минусы: Переписывает историю (не для shared веток).

Пример (предположим ветки main и feature):

```
# Merge
git checkout main
```

```
git merge feature  # Создает merge commit

# Rebase
git checkout feature
git rebase main    # Перебазирует feature на main
git checkout main
git merge feature  # Fast-forward
```

Ссылки: Atlassian — https://www.atlassian.com/git/tutorials/merging-vs-rebasing.

cherry pick, squash

- **cherry-pick**: Копирует конкретный коммит в текущую ветку. git cherry-pick .
- **squash**: Сливает несколько коммитов в один при merge/rebase. Используйте interactive rebase или merge --squash.

Пример:

```
git cherry-pick abc123  # Применяет коммит abc123

git rebase -i HEAD~3    # Interactive, измените pick на squash для слияния
```

Squash для чистоты истории. Ссылки: Git Docs — https://git-scm.com/docs/git-cherry-pick.

patch, stash

- **patch**: Применяет изменения из diff-файла. git apply patch.diff.
- **stash**: Временно сохраняет uncommitted изменения. git stash; git stash pop.

Пример:

```
git diff > changes.patch  # Создать patch
git apply changes.patch   # Применить

git stash                 # Сохранить
git stash list            # Список
git stash apply           # Применить без удаления
git stash pop             # Применить и удалить
```

Patch для обмена изменениями, stash для переключения веток. Ссылки: Baeldung — https://www.baeldung.com/git-stash-apply-pop.

reset, revert, с примерами наглядно с кодом

- **reset**: Перемещает HEAD, изменяет историю. Опасно для shared. --soft (staging), --mixed (unstage), --hard (discard).
- **revert**: Создает новый коммит, отменяющий изменения. Безопасно.

Примеры (предположим коммиты A-B-C, HEAD на C):

```
# Reset
git reset --hard B  # HEAD на B, удаляет C (файлы/история)

git reset --soft B  # HEAD на B, C в staging

git reset --mixed B # HEAD на B, C unstaged

# Revert
git revert C        # Создает D, отменяющий C. История A-B-C-D
```

Наглядно:
Исходная история: commit1 <- commit2 <- commit3 (HEAD)

После git reset --hard commit2: commit1 <- commit2 (HEAD) # commit3 потерян

После git revert commit3: commit1 <- commit2 <- commit3 <- revert-commit (HEAD) # Изменения commit3 отменены

Используйте revert для public, reset для local. Ссылки: Git Docs — https://git-scm.com/docs/git-reset; https://git-scm.com/docs/git-revert.

Модуль 4

Многопоточность

Многопоточность (multithreading) в Java — это возможность выполнения нескольких потоков (threads) одновременно в одном процессе, что позволяет эффективно использовать ресурсы CPU, обрабатывать параллельные задачи (например, UI и вычисления) и улучшать производительность. Потоки — это легковесные процессы внутри JVM. Многопоточность основана на пакете java.lang.Thread и java.util.concurrent. Принципы ООП здесь: абстракция (интерфейсы как Runnable), полиморфизм (разные реализации задач). Однако она вводит сложности как синхронизация. Java использует модель памяти (JMM — Java Memory Model) для видимости изменений между потоками. Подробнее: Oracle Docs — https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html.

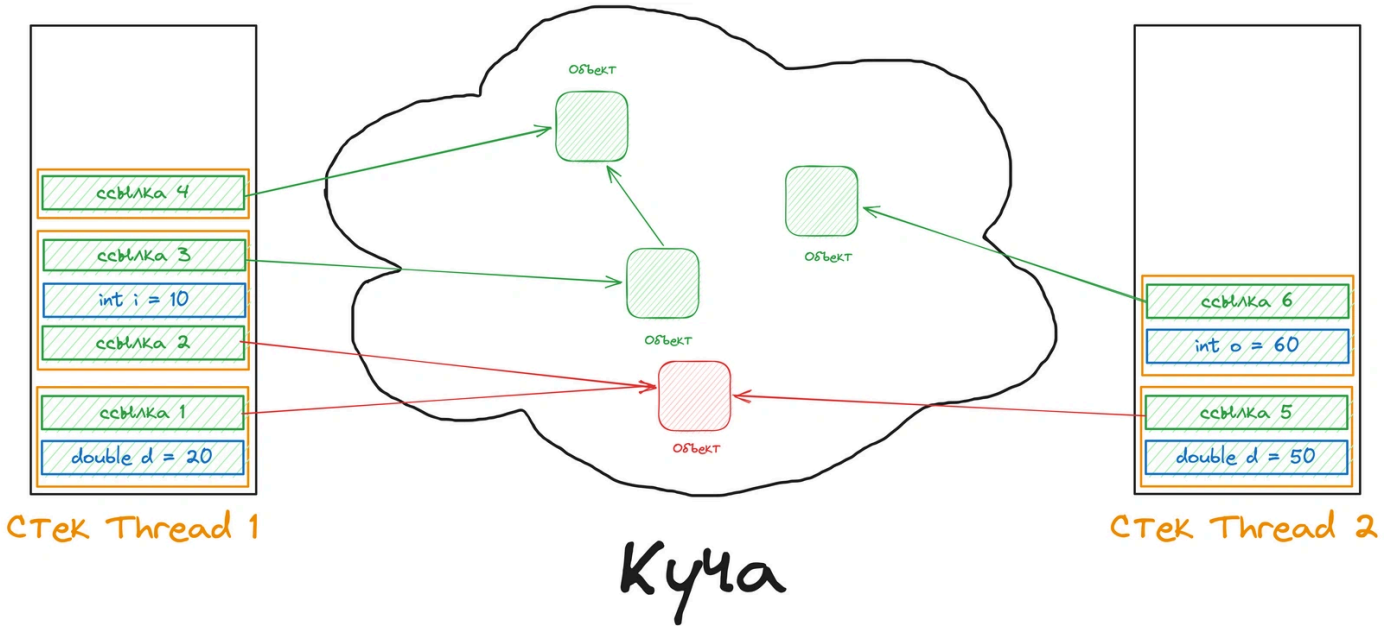
Влияние многопоточности в Java на память

Когда создаётся новый поток, для него выделяется отдельный фрейм стека. Это означает, что каждый поток имеет свой собственный стек вызовов, где хранятся его локальные переменные и

ссылки на объекты в куче. Эти стеки изолированы друг от друга, что исключает возможность взаимодействия потоков с локальными переменными друг друга.

Даже если два потока выполняют один и тот же код, они создают свои собственные локальные переменные в отдельных стеках. Таким образом, каждый поток имеет свою версию каждой локальной переменной.

В то же время все потоки совместно используют одну общую кучу, что означает, что объекты, созданные любым потоком, доступны для всех потоков. Это делает особенно важным управление доступом к объектам в куче, чтобы избежать проблем с согласованностью данных и условий гонки.



Для управления доступом нескольких потоков к общим ресурсам используется синхронизация. Она может быть реализована с помощью ключевого слова synchronized или через специальные классы, такие как ReentrantLock или Semaphore.

Сборщик мусора в Java также работает в многопоточной среде и способен обрабатывать объекты из всех потоков. Однако стоит уделять внимание долгоживущим объектам и ресурсам, которые могут блокировать работу сборщика мусора, ухудшая производительность системы.

Способы создания потоков

Потоки создаются для выполнения кода параллельно. Два основных способа:

1. **Наследование от Thread**: Переопределить run().
2. **Реализация интерфейса Runnable**: Предпочтительно, так как позволяет наследовать от других классов (гибкость ООП).

Пример реализации и использования:

```java
// Способ 1: Наследование от Thread
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Thread: " + i);
        }
    }
}

// Способ 2: Реализация Runnable
class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Runnable: " + i);
        }
    }
}

public class ThreadDemo {
    public static void main(String[] args) {
        // Использование Thread
        MyThread t1 = new MyThread();
        t1.start();  // Запуск потока (не run()!)

        // Использование Runnable
        Thread t2 = new Thread(new MyRunnable());
        t2.start();

        // Лямбда (с Java 8)
        Thread t3 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Lambda: " + i);
            }
        });
        t3.start();
    }
}
```

Вывод может быть перемешан из-за параллелизма. start() вызывает run() в новом потоке. Ссылки: Baeldung — https://www.baeldung.com/java-start-thread.

Виды состояния потоков

Потоки проходят жизненный цикл с состояниями (enum Thread.State):

- **NEW**: Создан, но не запущен (перед start()).
- **RUNNABLE**: Готов к выполнению (после start(), включает RUNNING когда CPU выделен).
- **BLOCKED**: Ждет монитора (lock) для synchronized.
- **WAITING**: Ждет уведомления (wait(), join() без timeout).
- **TIMED_WAITING**: Ждет с timeout (sleep(), wait(timeout)).
- **TERMINATED**: Завершен (run() закончен или исключение).

Пример использования:

```
public class StateDemo {
    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(() -> {
            try {
                Thread.sleep(1000);  // TIMED_WAITING
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        System.out.println(t.getState());  // NEW
        t.start();
        System.out.println(t.getState());  // RUNNABLE
        Thread.sleep(500);                 // Дать время на sleep
        System.out.println(t.getState());  // TIMED_WAITING
        t.join();                          // Ждать завершения
        System.out.println(t.getState());  // TERMINATED
    }
}
```

Это помогает в отладке. Ссылки: Oracle — https://docs.oracle.com/javase/21/docs/api/java/lang/Thread.State.html.

Ключевое слово volatile

Volatile — модификатор для полей, обеспечивающий видимость изменений между потоками (JMM). Запрещает кэширование в регистрах/кэше CPU, всегда читает/пишет в main memory. Не обеспечивает атомарность, только видимость и ordering.

Пример:

```
class VolatileDemo {
    private volatile boolean running = true;  // Volatile для видимости

    public void start() {
        new Thread(() -> {
            while (running) {
                // Бесконечный цикл
            }
            System.out.println("Stopped");
        }).start();
    }

    public void stop() {
        running = false;  // Изменение видно сразу
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        VolatileDemo demo = new VolatileDemo();
        demo.start();
        Thread.sleep(1000);
        demo.stop();  // Без volatile цикл может не остановиться
    }
}
```

Без volatile компилятор может оптимизировать и не увидеть изменения. Ссылки: Baeldung — https://www.baeldung.com/java-volatile.

Проблемы многопоточных приложений: состояние гонки, дедлоки и лайфлоки

- **Состояние гонки (Race Condition)**: Когда результат зависит от порядка выполнения потоков (например, инкремент shared переменной).
- **Дедлок (Deadlock)**: Потоки ждут друг друга (циклическая зависимость от locks).
- **Лайфлок (Livelock)**: Потоки активны, но не прогрессируют (например, вежливо уступают друг другу).

Пример race condition:

```
class Counter {
    private int count = 0;
```

```
    public void increment() {
        count++;  // Не атомарно: read-modify-write
    }

    public int getCount() { return count; }
}

public class RaceDemo {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Runnable task = () -> { for (int i = 0; i < 10000; i++)
counter.increment(); };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start(); t2.start();
        t1.join(); t2.join();
        System.out.println(counter.getCount());  // Может быть < 20000 из-за race
    }
}
```

Пример deadlock:

```
class DeadlockDemo {
    private final Object lock1 = new Object();
    private final Object lock2 = new Object();

    public void method1() {
        synchronized (lock1) {
            synchronized (lock2) {
                System.out.println("Method1");
            }
        }
    }

    public void method2() {
        synchronized (lock2) {
            synchronized (lock1) {
                System.out.println("Method2");
            }
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        DeadlockDemo demo = new DeadlockDemo();
        new Thread(demo::method1).start();
        new Thread(demo::method2).start();  // Дедлок: t1 держит lock1, ждет lock2; t2 наоборот
    }
}
```

Livelock: Потоки меняют состояние, но зацикливаются (редко, пример — два потока, уступающие resource). Решения: Синхронизация, locks. Ссылки: GeeksforGeeks — https://www.geeksforgeeks.org/deadlock-starvation-and-livelock/.

Пакет java.util.concurrent

Пакет предоставляет high-level API для concurrency (с Java 5), включая executors, queues, synchronizers. Упрощает многопоточность, избегает low-level Thread. Включает Atomic, Lock, Executors, Future и т.д. Поддерживает полиморфизм через интерфейсы. Ссылки: Oracle — https://docs.oracle.com/javase/21/docs/api/java/util/concurrent/package-summary.html.

Atomic

Atomic-классы (как AtomicInteger) обеспечивают атомарные операции без locks (CAS — Compare-And-Swap). Для thread-safe модификаций.

Пример:

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger count = new AtomicInteger(0);

    public void increment() {
        count.incrementAndGet();  // Атомарно
    }

    public int getCount() { return count.get(); }
}

public class AtomicDemo {
    public static void main(String[] args) throws InterruptedException {
        AtomicCounter counter = new AtomicCounter();
        Runnable task = () -> { for (int i = 0; i < 10000; i++)
counter.increment(); };
```

```
        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start(); t2.start();
        t1.join(); t2.join();
        System.out.println(counter.getCount());  // Всегда 20000
    }
}
```

Другие: AtomicBoolean, AtomicReference. Ссылки: Baeldung — https://www.baeldung.com/java-atomic-variables.

Lock

Интерфейс Lock (ReentrantLock реализация) — альтернатива synchronized, более гибкий (tryLock, lockInterruptibly).

Пример:

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class LockCounter {
    private int count = 0;
    private Lock lock = new ReentrantLock();

    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();  // Всегда в finally
        }
    }

    public int getCount() { return count; }
}

public class LockDemo {
    // Аналогично AtomicDemo, но с lock — count всегда 20000
}
```

Плюсы: Fairness (new ReentrantLock(true)), условия (Condition). Ссылки: Oracle — https://docs.oracle.com/javase/21/docs/api/java/util/concurrent/locks/Lock.html.

Executors

Executors — фабрика для создания ExecutorService (пулы потоков). Упрощает управление потоками.

Пример:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorsDemo {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);  // Пул из 2 потоков

        for (int i = 0; i < 5; i++) {
            int taskId = i;
            executor.submit(() -> {
                System.out.println("Task " + taskId + " executed by " + Thread.currentThread().getName());
            });
        }

        executor.shutdown();  // Завершить после задач
    }
}
```

Виды: newSingleThreadExecutor, newCachedThreadPool. Ссылки: Baeldung — https://www.baeldung.com/java-executor-service-tutorial.

Future, CompletableFuture

- **Future**: Представляет результат асинхронной задачи (get() блокирует).
- **CompletableFuture**: Более мощный (Java 8), цепочки, комбинации, обработка ошибок.

Пример Future:

```
import java.util.concurrent.*;

public class FutureDemo {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        Future<Integer> future = executor.submit(() -> {
```

```
            Thread.sleep(1000);
            return 42;
        });

        System.out.println("Doing other work...");
        Integer result = future.get();  // Блокирует до готовности
        System.out.println("Result: " + result);
        executor.shutdown();
    }
}
```

Пример CompletableFuture:

```
import java.util.concurrent.CompletableFuture;

public class CompletableFutureDemo {
    public static void main(String[] args) {
        CompletableFuture<String> cf = CompletableFuture.supplyAsync(() -> {
            try { Thread.sleep(1000); } catch (InterruptedException e) {}
            return "Hello";
        }).thenApply(s -> s + " World")  // Цепочка
          .thenApply(String::toUpperCase);

        System.out.println("Doing other work...");
        System.out.println(cf.join());  // HELLO WORLD
    }
}
```

CompletableFuture для non-blocking, combine(). Ссылки: Baeldung — https://www.baeldung.com/java-completablefuture.

Синхронизаторы

Синхронизаторы — классы для координации потоков: CountDownLatch, CyclicBarrier, Semaphore, Phaser, Exchanger.

- **CountDownLatch**: Ждет, пока N потоков не завершат (countDown()).
- **CyclicBarrier**: Потоки ждут друг друга на барьере (reusable).
- **Semaphore**: Контролирует доступ (permits, как mutex).

Пример CountDownLatch:

```
import java.util.concurrent.CountDownLatch;
```

```
public class LatchDemo {
    public static void main(String[] args) throws InterruptedException {
        CountDownLatch latch = new CountDownLatch(3);  // Ждать 3

        for (int i = 0; i < 3; i++) {
            new Thread(() -> {
                System.out.println("Working...");
                latch.countDown();  // Уменьшить счетчик
            }).start();
        }

        latch.await();  // Ждать до 0
        System.out.println("All done!");
    }
}
```

Пример Semaphore:

```
import java.util.concurrent.Semaphore;

public class SemaphoreDemo {
    private static Semaphore semaphore = new Semaphore(2);  // 2 permits

    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            new Thread(() -> {
                try {
                    semaphore.acquire();  // Получить permit
                    System.out.println(Thread.currentThread().getName() + "
working");
                    Thread.sleep(1000);
                } catch (InterruptedException e) {} finally {
                    semaphore.release();  // Освободить
                }
            }).start();
        }
    }
}
```

Только 2 потока одновременно. Ссылки: Oracle — https://docs.oracle.com/javase/21/docs/api/java/util/concurrent/package-summary.html (синхронизаторы в пакете).

Модуль 5

SOLID

SOLID — это акроним для пяти принципов проектирования объектно-ориентированного программирования, которые помогают создавать масштабируемые, поддерживаемые и гибкие системы. Эти принципы, предложенные Робертом Мартином, применяются в Java для улучшения архитектуры кода. Они основаны на ООП и минимизируют зависимости, улучшая читаемость и тестируемость.

- **S — Single Responsibility Principle (Принцип единственной ответственности)**: Класс должен иметь только одну причину для изменения (одна ответственность). Пример: Класс для обработки заказов не должен заниматься логированием.
- **O — Open/Closed Principle (Принцип открытости/закрытости)**: Классы должны быть открыты для расширения, но закрыты для модификации. Используйте абстракции (интерфейсы, абстрактные классы). Пример: Добавление новой функциональности через наследование, а не изменение кода.
- **L — Liskov Substitution Principle (Принцип подстановки Барбары Лисков)**: Объекты подкласса должны быть заменяемы объектами суперкласса без нарушения поведения. Пример: Если Square extends Rectangle, вызов setWidth() не должен ломать логику.
- **I — Interface Segregation Principle (Принцип разделения интерфейсов)**: Клиенты не должны зависеть от интерфейсов, которые не используют. Пример: Разделить большой интерфейс на меньшие, специфичные.
- **D — Dependency Inversion Principle (Принцип инверсии зависимостей)**: Высокоуровневые модули не должны зависеть от низкоуровневых; оба зависят от абстракций. Пример: Использовать Dependency Injection.

Пример кода (Single Responsibility):

```
// Плохо: Один класс делает всё
class Order {
    void processOrder() { /* обработка */ }
    void saveToDatabase() { /* сохранение */ }
    void sendEmail() { /* отправка email */ }
}

// Хорошо: Разделение
class OrderProcessor {
    void processOrder() { /* обработка */ }
}

class OrderRepository {
    void saveToDatabase() { /* сохранение */ }
}
```

```
class EmailService {
    void sendEmail() { /* отправка email */ }
}
```

Ссылки: Baeldung — https://www.baeldung.com/solid-principles.

Принципы SOLID в Java с наглядным примером

SOLID — это акроним, обозначающий пять принципов проектирования объектно-ориентированного программирования, сформулированных Робертом Мартином (Uncle Bob). Эти принципы помогают создавать масштабируемый, поддерживаемый и гибкий код. Я подробно разберу каждый принцип, объясню его назначение, плюсы/минусы, и продемонстрирую их на едином примере — системе управления заказами в интернет-магазине. Каждый принцип будет показан сначала с нарушением, затем с правильной реализацией. Код будет на Java, с комментариями и объяснениями. В конце — ссылки на достоверные источники.

Принципы SOLID

1. **S** — Single Responsibility Principle (Принцип единственной ответственности): Класс должен иметь только одну причину для изменения, т.е. выполнять одну задачу.
2. **O** — Open/Closed Principle (Принцип открытости/закрытости): Классы должны быть открыты для расширения, но закрыты для модификации.
3. **L** — Liskov Substitution Principle (Принцип подстановки Барбары Лисков): Объекты подкласса должны быть взаимозаменяемы с объектами родительского класса без нарушения поведения программы.
4. **I** — Interface Segregation Principle (Принцип разделения интерфейса): Клиенты не должны зависеть от интерфейсов, которые они не используют.
5. **D** — Dependency Inversion Principle (Принцип инверсии зависимостей): Модули высокого уровня не должны зависеть от модулей низкого уровня. Оба должны зависеть от абстракций, а абстракции — не зависеть от деталей.

Практический пример: Система управления заказами

Сценарий: Создаем систему для интернет-магазина, которая обрабатывает заказы, отправляет уведомления и сохраняет данные. Мы начнем с кода, нарушающего SOLID, затем переработаем его, чтобы соответствовать каждому принципу.

Нарушение SOLID (плохой код)

Этот код нарушает все принципы SOLID. Класс `OrderProcessor` делает слишком много: создает заказы, отправляет email, сохраняет в базу, рассчитывает скидки.

```
// Плохой код: нарушает SOLID
class Order {
    String customerEmail;
    double price;
    String item;

    Order(String customerEmail, double price, String item) {
        this.customerEmail = customerEmail;
        this.price = price;
        this.item = item;
    }
}

class OrderProcessor {
    // Нарушение SRP: класс делает всё
    public void processOrder(Order order) {
        // Логика обработки заказа
        System.out.println("Processing order for " + order.item);

        // Сохранение в базу (прямая зависимость от SQL)
        System.out.println("Saving to SQL DB: " + order.customerEmail + ", " +
order.price);

        // Отправка email
        System.out.println("Sending email to " + order.customerEmail);

        // Расчет скидки (только для VIP)
        if (order.customerEmail.contains("vip")) {
            order.price *= 0.9; // 10% скидка
            System.out.println("VIP discount applied, new price: " + order.price);
        }
    }
}

public class BadOrderSystem {
    public static void main(String[] args) {
        Order order = new Order("vip@example.com", 100.0, "Laptop");
        OrderProcessor processor = new OrderProcessor();
        processor.processOrder(order);
    }
}
```

Проблемы:

- **SRP**: `OrderProcessor` отвечает за обработку, сохранение, отправку email и скидки.
- **OCP**: Добавление нового типа уведомления (SMS) или базы (NoSQL) требует изменения кода.
- **LSP**: Нет подклассов, но если бы были (например, `VipOrder`), могли бы нарушить поведение.
- **ISP**: Нет интерфейсов, но если бы были, клиенты могли бы зависеть от ненужных методов.
- **DIP**: Прямая зависимость от SQL и email.

Переработка кода по принципам SOLID

Теперь перепишем код, чтобы он соответствовал всем принципам SOLID. Мы создадим модульную систему с интерфейсами, абстракциями и правильным разделением ответственности.

1. Single Responsibility Principle (SRP)

Описание: Класс должен иметь одну причину для изменения. Если класс делает несколько вещей (например, обработка заказа и отправка email), его сложно поддерживать.

Нарушение в плохом коде: `OrderProcessor` отвечает за обработку, сохранение, уведомления и скидки.

Исправление: Разделим ответственность:

- `OrderProcessor`: Только обработка заказа.
- `OrderRepository`: Сохранение данных.
- `NotificationService`: Отправка уведомлений.
- `DiscountCalculator`: Расчет скидок.

Код:

```
// Класс заказа (POJO)
class Order {
    private String customerEmail;
    private double price;
    private String item;

    public Order(String customerEmail, double price, String item) {
        this.customerEmail = customerEmail;
        this.price = price;
        this.item = item;
    }

    public String getCustomerEmail() { return customerEmail; }
    public double getPrice() { return price; }
    public String getItem() { return item; }
```

```
    public void setPrice(double price) { this.price = price; }
}

// Интерфейс для сохранения
interface OrderRepository {
    void save(Order order);
}

// Реализация для SQL
class SqlOrderRepository implements OrderRepository {
    @Override
    public void save(Order order) {
        System.out.println("Saving to SQL DB: " + order.getCustomerEmail() + ", " +
order.getPrice());
    }
}

// Интерфейс для уведомлений
interface NotificationService {
    void notify(Order order);
}

// Реализация для email
class EmailNotificationService implements NotificationService {
    @Override
    public void notify(Order order) {
        System.out.println("Sending email to " + order.getCustomerEmail());
    }
}

// Интерфейс для скидок
interface DiscountCalculator {
    double calculateDiscount(Order order);
}

// Реализация для VIP
class VipDiscountCalculator implements DiscountCalculator {
    @Override
    public double calculateDiscount(Order order) {
        if (order.getCustomerEmail().contains("vip")) {
            return order.getPrice() * 0.9; // 10% скидка
        }
        return order.getPrice();
```

```
    }
}

// Обработка заказа
class OrderProcessor {
    private final OrderRepository repository;
    private final NotificationService notificationService;
    private final DiscountCalculator discountCalculator;

    public OrderProcessor(OrderRepository repository, NotificationService
notificationService,
                          DiscountCalculator discountCalculator) {
        this.repository = repository;
        this.notificationService = notificationService;
        this.discountCalculator = discountCalculator;
    }

    public void processOrder(Order order) {
        System.out.println("Processing order for " + order.getItem());
        order.setPrice(discountCalculator.calculateDiscount(order));
        repository.save(order);
        notificationService.notify(order);
    }
}
```

Объяснение:

- Каждый класс имеет одну ответственность:
  - `OrderProcessor`: Координирует процесс.
  - `SqlOrderRepository`: Сохранение.
  - `EmailNotificationService`: Уведомления.
  - `VipDiscountCalculator`: Скидки.
- **Плюсы**: Изменение одной функции (например, способа уведомления) не затрагивает другие классы.
- **Тест**: Добавьте `NoSqlOrderRepository` — не нужно менять `OrderProcessor`.

---

2. Open/Closed Principle (OCP)

Описание: Классы должны быть открыты для расширения (новые реализации), но закрыты для модификации (без изменения исходного кода).

Нарушение в плохом коде: Добавление SMS-уведомлений или NoSQL требует изменения `OrderProcessor`.

Исправление: Используем интерфейсы (`OrderRepository`, `NotificationService`, `DiscountCalculator`), чтобы новые реализации добавлялись без изменения существующего кода.

Код (дополнение к предыдущему):

```
// Новая реализация: SMS уведомления
class SmsNotificationService implements NotificationService {
    @Override
    public void notify(Order order) {
        System.out.println("Sending SMS to " + order.getCustomerEmail());
    }
}

// Новая реализация: NoSQL база
class NoSqlOrderRepository implements OrderRepository {
    @Override
    public void save(Order order) {
        System.out.println("Saving to NoSQL DB: " + order.getCustomerEmail() + ", "
+ order.getPrice());
    }
}

// Новая реализация: Сезонная скидка
class SeasonalDiscountCalculator implements DiscountCalculator {
    @Override
    public double calculateDiscount(Order order) {
        return order.getPrice() * 0.85; // 15% скидка для всех
    }
}
```

Демонстрация:

```
public class SolidOrderSystem {
    public static void main(String[] args) {
        // Конфигурация
        OrderRepository sqlRepo = new SqlOrderRepository();
        NotificationService emailService = new EmailNotificationService();
        DiscountCalculator vipDiscount = new VipDiscountCalculator();

        OrderProcessor processor = new OrderProcessor(sqlRepo, emailService,
vipDiscount);
```

```
        Order order = new Order("vip@example.com", 100.0, "Laptop");
        processor.processOrder(order);

        // Расширение: SMS + NoSQL + сезонная скидка
        OrderProcessor newProcessor = new OrderProcessor(
            new NoSqlOrderRepository(),
            new SmsNotificationService(),
            new SeasonalDiscountCalculator()
        );
        newProcessor.processOrder(order);
    }
}
```

Ожидаемый вывод:

```
Processing order for Laptop
Saving to SQL DB: vip@example.com, 90.0
Sending email to vip@example.com
Processing order for Laptop
Saving to NoSQL DB: vip@example.com, 85.0
Sending SMS to vip@example.com
```

Объяснение:

- **Открытость**: Добавлены `SmsNotificationService`, `NoSqlOrderRepository`, `SeasonalDiscountCalculator` без изменения `OrderProcessor`.
- **Закрытость**: `OrderProcessor` не меняется, работает с абстракциями.
- **Плюсы**: Легко добавить новый тип уведомления (например, Push) или базы (MongoDB).

---

3. Liskov Substitution Principle (LSP)

Описание: Подклассы должны быть взаимозаменяемы с родительским классом без изменения поведения программы. Поведение не должно ломаться при замене.

Нарушение (гипотетическое):

```
class VipOrder extends Order {
    public VipOrder(String customerEmail, double price, String item) {
        super(customerEmail, price, item);
    }

    // Нарушение: метод выбрасывает исключение, чего не делает базовый
    @Override
    public double getPrice() {
        throw new UnsupportedOperationException("VIP prices are secret");
```

```
    }
}
```

Проблема: Если `OrderProcessor` ожидает `Order.getPrice()`, то `VipOrder` сломает логику.

Исправление: Гарантируем, что подклассы (например, `VipOrder`) не нарушают контракт `Order`.

Код:

```
class VipOrder extends Order {
    public VipOrder(String customerEmail, double price, String item) {
        super(customerEmail, price, item);
    }

    // Корректное поведение: добавляем функциональность, не ломая контракт
    public double getVipBonus() {
        return getPrice() * 0.05; // Бонус 5%
    }
}

// Тест
public class LspDemo {
    public static void main(String[] args) {
        OrderProcessor processor = new OrderProcessor(
            new SqlOrderRepository(),
            new EmailNotificationService(),
            new VipDiscountCalculator()
        );

        // Работает с Order
        Order order = new Order("user@example.com", 100.0, "Laptop");
        processor.processOrder(order);

        // Работает с VipOrder (LSP)
        Order vipOrder = new VipOrder("vip@example.com", 100.0, "Phone");
        processor.processOrder(vipOrder); // Без ошибок
    }
}
```

Объяснение:

- `VipOrder` расширяет `Order`, не ломая `getPrice()` или другие методы.
- `OrderProcessor` работает с любым `Order` (включая `VipOrder`).
- **Плюсы**: Заменяемость гарантирует, что новые типы заказов не сломают систему.

---

4. Interface Segregation Principle (ISP)

Описание: Клиенты не должны зависеть от интерфейсов, которые они не используют. Интерфейсы должны быть узкими и специфичными.

Нарушение (гипотетическое):

```
interface OrderOperations {
    void save(Order order);
    void notify(Order order);
    double calculateDiscount(Order order);
}

class OrderProcessor implements OrderOperations {
    // Должен реализовать все методы, даже если не все нужны
}
```

Проблема: `OrderProcessor` вынужден реализовать ненужные методы (например, сохранение).

Исправление: Разделяем интерфейсы на `OrderRepository`, `NotificationService`, `DiscountCalculator` (как выше). Каждый клиент реализует только нужное.

Код (уже соответствует):

- `OrderProcessor` зависит от трех узких интерфейсов, каждый с одной задачей.
- Если добавить `PushNotificationService`, он реализует только `NotificationService`.

Тест:

```
// Добавим Push уведомления
class PushNotificationService implements NotificationService {
    @Override
    public void notify(Order order) {
        System.out.println("Sending push notification to " +
order.getCustomerEmail());
    }
}

public class IspDemo {
    public static void main(String[] args) {
        OrderProcessor processor = new OrderProcessor(
            new SqlOrderRepository(),
            new PushNotificationService(), // Только notify
            new VipDiscountCalculator()
        );
```

```
        processor.processOrder(new Order("user@example.com", 100.0, "Tablet"));
    }
}
```

Объяснение:

- `PushNotificationService` реализует только `notify`, не заботясь о других операциях.
- **Плюсы**: Классы не содержат лишнего кода, легче тестировать.

---

5. Dependency Inversion Principle (DIP)

Описание: Высокоуровневые модули (OrderProcessor) не должны зависеть от низкоуровневых (SqlOrderRepository). Оба зависят от абстракций (интерфейсов).

Нарушение в плохом коде: `OrderProcessor` напрямую использует SQL и email-логику.

Исправление: Зависимости передаются через конструктор (Dependency Injection), а `OrderProcessor` работает с интерфейсами.

Код (уже соответствует):

- `OrderProcessor` принимает `OrderRepository`, `NotificationService`, `DiscountCalculator` через конструктор.
- Реализации (SqlOrderRepository, EmailNotificationService) — низкоуровневые модули, зависят от тех же интерфейсов.

Тест с DI:

```
public class DipDemo {
    public static void main(String[] args) {
        // Внедрение зависимостей
        OrderRepository repo = new NoSqlOrderRepository();
        NotificationService notification = new EmailNotificationService();
        DiscountCalculator discount = new SeasonalDiscountCalculator();

        OrderProcessor processor = new OrderProcessor(repo, notification,
discount);
        processor.processOrder(new Order("user@example.com", 100.0, "Book"));
    }
}
```

Объяснение:

- `OrderProcessor` не знает о конкретных реализациях (SQL/NoSQL, Email/SMS).
- Зависимости внедряются извне (например, через Spring DI).

- **Плюсы**: Легко заменить реализацию (например, на `MockRepository` для тестов).

Полный код (соответствует SOLID)

```
import java.util.*;

// Класс заказа
class Order {
    private String customerEmail;
    private double price;
    private String item;

    public Order(String customerEmail, double price, String item) {
        this.customerEmail = customerEmail;
        this.price = price;
        this.item = item;
    }

    public String getCustomerEmail() { return customerEmail; }
    public double getPrice() { return price; }
    public String getItem() { return item; }
    public void setPrice(double price) { this.price = price; }
}

// Интерфейсы
interface OrderRepository {
    void save(Order order);
}

interface NotificationService {
    void notify(Order order);
}

interface DiscountCalculator {
    double calculateDiscount(Order order);
}

// Реализации
class SqlOrderRepository implements OrderRepository {
    @Override
    public void save(Order order) {
        System.out.println("Saving to SQL DB: " + order.getCustomerEmail() + ", " +
order.getPrice());
```

```
    }
}

class NoSqlOrderRepository implements OrderRepository {
    @Override
    public void save(Order order) {
        System.out.println("Saving to NoSQL DB: " + order.getCustomerEmail() + ", "
+ order.getPrice());
    }
}

class EmailNotificationService implements NotificationService {
    @Override
    public void notify(Order order) {
        System.out.println("Sending email to " + order.getCustomerEmail());
    }
}

class SmsNotificationService implements NotificationService {
    @Override
    public void notify(Order order) {
        System.out.println("Sending SMS to " + order.getCustomerEmail());
    }
}

class VipDiscountCalculator implements DiscountCalculator {
    @Override
    public double calculateDiscount(Order order) {
        if (order.getCustomerEmail().contains("vip")) {
            return order.getPrice() * 0.9;
        }
        return order.getPrice();
    }
}

class SeasonalDiscountCalculator implements DiscountCalculator {
    @Override
    public double calculateDiscount(Order order) {
        return order.getPrice() * 0.85;
    }
}

class OrderProcessor {
```

```
    private final OrderRepository repository;
    private final NotificationService notificationService;
    private final DiscountCalculator discountCalculator;

    public OrderProcessor(OrderRepository repository, NotificationService
notificationService,
                          DiscountCalculator discountCalculator) {
        this.repository = repository;
        this.notificationService = notificationService;
        this.discountCalculator = discountCalculator;
    }

    public void processOrder(Order order) {
        System.out.println("Processing order for " + order.getItem());
        order.setPrice(discountCalculator.calculateDiscount(order));
        repository.save(order);
        notificationService.notify(order);
    }
}

// Демонстрация
public class SolidOrderSystem {
    public static void main(String[] args) {
        // Конфигурация 1: SQL + Email + VIP
        OrderProcessor processor1 = new OrderProcessor(
            new SqlOrderRepository(),
            new EmailNotificationService(),
            new VipDiscountCalculator()
        );
        Order order1 = new Order("vip@example.com", 100.0, "Laptop");
        processor1.processOrder(order1);

        System.out.println("---");

        // Конфигурация 2: NoSQL + SMS + Seasonal
        OrderProcessor processor2 = new OrderProcessor(
            new NoSqlOrderRepository(),
            new SmsNotificationService(),
            new SeasonalDiscountCalculator()
        );
        Order order2 = new Order("user@example.com", 100.0, "Phone");
        processor2.processOrder(order2);
```

```
    }
}
```

Подробное объяснение кода

1. **SRP**:
   - Каждый класс/интерфейс отвечает за одну задачу.
   - `OrderProcessor` координирует, но не реализует детали (делегирует).
   - Изменение базы данных не затрагивает уведомления.
2. **OCP**:
   - Интерфейсы позволяют добавлять новые реализации (`SmsNotificationService`, `NoSqlOrderRepository`) без изменения `OrderProcessor`.
   - Пример расширения: `PushNotificationService` или `MongoDbRepository`.
3. **LSP**:
   - `VipOrder` (если добавить) не ломает контракт `Order`.
   - `OrderProcessor` работает с любым `Order`.
4. **ISP**:
   - Узкие интерфейсы (`OrderRepository`, `NotificationService`) минимизируют зависимости.
   - `EmailNotificationService` не знает о сохранении или скидках.
5. **DIP**:
   - `OrderProcessor` зависит от абстракций (интерфейсов), а не от конкретных классов.
   - Внедрение через конструктор упрощает тестирование и замену.

Плюсы и минусы применения SOLID

Плюсы:

- **Масштабируемость**: Легко добавлять новые типы (например, Push-уведомления).
- **Поддерживаемость**: Изменение одной функции не ломает другие.
- **Тестируемость**: Узкие интерфейсы и DI упрощают mock-тесты.
- **Гибкость**: Легко заменить реализацию (SQL → NoSQL).

Минусы:

- **Сложность**: Больше классов и интерфейсов (в примере 10+ классов вместо 2).
- **Overhead**: Требуется больше времени на проектирование.
- **Избыточность**: Для маленьких проектов SOLID может быть избыточным.

Тестирование и проверка

1. **Добавьте новый тип уведомления**:

```
class PushNotificationService implements NotificationService {
    @Override
    public void notify(Order order) {
        System.out.println("Sending push to " + order.getCustomerEmail());
    }
}
```

   Замените `EmailNotificationService` на `PushNotificationService` в `main`.

2. **Добавьте новый тип заказа**:

```
class BulkOrder extends Order {
    public BulkOrder(String customerEmail, double price, String item) {
        super(customerEmail, price, item);
    }
}
```

   Проверьте, что `OrderProcessor` работает с `BulkOrder`.

3. **Тест производительности**: Для маленькой системы SOLID может добавить overhead, но для больших проектов экономит время поддержки.

Ссылки на достоверные источники

- Robert Martin's Clean Code (SOLID Principles): https://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882
- Baeldung (SOLID in Java): https://www.baeldung.com/solid-principles
- Refactoring.Guru (SOLID): https://refactoring.guru/design-patterns/solid-principles
- Oracle Java Tutorials (OOP Concepts): https://docs.oracle.com/javase/tutorial/java/concepts/
- GeeksforGeeks (SOLID): https://www.geeksforgeeks.org/solid-principles-in-java/
- Medium (SOLID with Examples): https://medium.com/@aarkay0106/solid-principles-in-java-with-real-world-examples-7b8c2f6d7a1f

---

Заключение

Этот пример демонстрирует, как SOLID делает код модульным, расширяемым и легким для поддержки. Нарушение принципов (плохой код) приводит к хрупкости и трудностям масштабирования. Применение SOLID требует больше кода, но окупается в долгосрочной перспективе.

Если нужны дополнительные примеры (например, конкретный принцип в другом контексте, тесты, или интеграция с фреймворками вроде Spring), уточните!

```
# Паттерны проектирования

Паттерны проектирования — это проверенные решения типичных задач проектирования. Они классифицируются на три группы: порождающие, структурные, поведенческие. В Java они широко используются для упрощения архитектуры, повторного использования кода и соблюдения SOLID.

- **Порождающие** (создание объектов):
  - **Singleton**: Один экземпляр класса. Пример: Logger.
  - **Factory Method**: Делегирует создание объектов подклассам.
  - **Abstract Factory**: Создает семейства связанных объектов.
  - **Builder**: Пошаговое создание сложных объектов. Пример: StringBuilder.

- **Структурные** (организация классов/объектов):
  - **Adapter**: Преобразует интерфейс одного класса в другой. Пример: InputStreamReader.
  - **Decorator**: Динамически добавляет функциональность. Пример: BufferedReader.
  - **Facade**: Упрощает доступ к сложной системе. Пример: JDBC.

- **Поведенческие** (взаимодействие объектов):
  - **Observer**: Один ко многим, уведомления об изменениях. Пример: Java’s Observable.
  - **Strategy**: Выбор алгоритма на лету. Пример: Comparator.
  - **Command**: Инкапсулирует запрос как объект. Пример: Runnable.

Пример Singleton:
```java
public class Singleton {
    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}  // Приватный конструктор

    public static Singleton getInstance() {
        return INSTANCE;
    }

    public void doSomething() {
        System.out.println("Singleton working");
    }
```
```

```
}

public class SingletonDemo {
    public static void main(String[] args) {
        Singleton singleton = Singleton.getInstance();
        singleton.doSomething();
    }
}
```

Ссылки: Refactoring.Guru — https://refactoring.guru/design-patterns; Oracle — https://docs.oracle.com/javase/tutorial/java/generics/index.html (паттерны в generics).

Web

Веб-разработка в Java включает серверные приложения (backend) с использованием Servlets, Spring, или Jakarta EE. Это сервер-клиентская модель: клиент (браузер) отправляет запросы (HTTP), сервер обрабатывает и отвечает. Java подходит для масштабируемых enterprise-приложений. Принципы ООП: абстракция (интерфейсы HTTP), инкапсуляция (логика сервера). Основные протоколы — HTTP/HTTPS. Ссылки: Baeldung — https://www.baeldung.com/java-web-overview.

Сетевая модель OSI (поверхностно)

Модель OSI (Open Systems Interconnection) — концептуальная модель для стандартизации сетевых взаимодействий. Состоит из 7 уровней:

1. **Физический**: Передача битов (кабели, сигналы).
2. **Канальный**: Надежная передача между узлами (Ethernet).
3. **Сетевой**: Маршрутизация (IP).
4. **Транспортный**: Надежная доставка (TCP, UDP).
5. **Сеансовый**: Управление сессиями.
6. **Представления**: Формат данных (шифрование, сжатие).
7. **Прикладной**: Интерфейс для приложений (HTTP, FTP).

Java работает на уровнях 4–7 (TCP, HTTP). Пример: Socket (транспортный), Servlets (прикладной). Ссылки: GeeksforGeeks — https://www.geeksforgeeks.org/layers-of-osi-model/.

Socket

Socket — это endpoint для сетевого взаимодействия (TCP/UDP). Java предоставляет java.net.Socket (клиент) и java.net.ServerSocket (сервер) для низкоуровневого сетевого программирования. Используется для клиент-серверного общения, например, чатов.

Пример клиент-сервер:

```
import java.io.*;
import java.net.*;

// Сервер
class Server {
    public static void main(String[] args) throws IOException {
        try (ServerSocket server = new ServerSocket(1234)) {
            System.out.println("Server started");
            try (Socket client = server.accept();
                 PrintWriter out = new PrintWriter(client.getOutputStream(), true);
                 BufferedReader in = new BufferedReader(new
InputStreamReader(client.getInputStream()))) {
                String input = in.readLine();
                out.println("Server received: " + input);
            }
        }
    }
}

// Клиент
class Client {
    public static void main(String[] args) throws IOException {
        try (Socket socket = new Socket("localhost", 1234);
             PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
             BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()))) {
            out.println("Hello from client");
            System.out.println("Client received: " + in.readLine());
        }
    }
}
```

Запустите Server, затем Client. Ссылки: Oracle — https://docs.oracle.com/javase/21/docs/api/java/net/Socket.html.

Протокол HTTP

HTTP (HyperText Transfer Protocol) — протокол прикладного уровня для передачи данных (веб). Базируется на запрос-ответ (request-response). Методы: GET, POST, PUT, DELETE. Статусы: 200 (OK), 404 (Not Found), 500 (Server Error). В Java обрабатывается через HttpClient (Java 11+) или Servlets.

Пример HttpClient:

```
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;

public class HttpClientDemo {
    public static void main(String[] args) throws Exception {
        HttpClient client = HttpClient.newHttpClient();
        HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create("https://api.example.com/data"))
                .GET()
                .build();
        HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
        System.out.println("Status: " + response.statusCode());
        System.out.println("Body: " + response.body());
    }
}
```

Ссылки: Oracle — https://docs.oracle.com/en/java/javase/21/docs/api/java.net.http/java/net/http/HttpClient.html.

REST

REST (Representational State Transfer) — архитектурный стиль для веб-API. Использует HTTP, ресурсы (URI), методы (GET/POST), JSON/XML. Принципы: stateless, client-server, uniform interface. Java реализует через JAX-RS (Jakarta RESTful Web Services) или Spring REST.

Пример JAX-RS (с Jakarta EE):

```
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

@Path("/hello")
public class HelloResource {
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String sayHello() {
        return "Hello, REST!";
```

```
    }
}
```

Развернуть на сервере (например, Tomcat). Доступ: http://localhost:8080/api/hello. Ссылки: Baeldung — https://www.baeldung.com/jax-rs-spec-and-implementations.

Servlets, Apache Tomcat

Servlets — Java-классы для обработки HTTP-запросов в веб-приложениях. Работают на сервере (контейнер сервлетов, как Apache Tomcat). Tomcat — популярный сервер, реализующий спецификации Servlet и JSP. Servlets — основа для веб-приложений, поддерживают REST, MVC. Это реализация прикладного уровня OSI, инкапсуляция логики обработки запросов.

Основы Servlets

- Servlet — класс, реализующий javax.servlet.Servlet (или extends HttpServlet).
- Жизненный цикл: init(), service(), destroy().
- Обработка: doGet(), doPost(), doPut(), etc.
- Конфигурация: web.xml или аннотации (@WebServlet).

Установка Apache Tomcat

1. Скачать с https://tomcat.apache.org/ (например, Tomcat 10 для Jakarta EE).
2. Распаковать, запустить bin/startup.sh (Unix) или startup.bat (Windows).
3. Доступ: http://localhost:8080.

Пример проекта с Servlet

Создадим веб-приложение с сервлетом, обрабатывающим GET/POST.

1. **Структура проекта** (Maven):

```
my-web-app/
├── pom.xml
└── src/
    └── main/
        ├── java/
        │   └── com/example/
        │       └── HelloServlet.java
        └── webapp/
            └── WEB-INF/
                └── web.xml
```

2. **POM-файл** (pom.xml):

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>my-web-app</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <dependencies>
        <dependency>
            <groupId>jakarta.servlet</groupId>
            <artifactId>jakarta.servlet-api</artifactId>
            <version>6.0.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <version>3.4.0</version>
            </plugin>
        </plugins>
    </build>
</project>
```

3. **Servlet** (src/main/java/com/example/HelloServlet.java):

```java
package com.example;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

import java.io.IOException;
import java.io.PrintWriter;

@WebServlet("/hello")
```

```
public class HelloServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter();
        out.println("<h1>Hello, GET!</h1>");
        out.println("<form method='post' action='hello'><input type='text'
name='name'><input type='submit'></form>");
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        String name = req.getParameter("name");
        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter();
        out.println("<h1>Hello, " + (name != null ? name : "POST") + "!</h1>");
    }

    @Override
    public void init() throws ServletException {
        System.out.println("Servlet initialized");
    }

    @Override
    public void destroy() {
        System.out.println("Servlet destroyed");
    }
}
```

4. **web.xml** (src/main/webapp/WEB-INF/web.xml, опционально, если не использовать @WebServlet):

```
<web-app xmlns="https://jakarta.ee/xml/ns/jakartaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
https://jakarta.ee/xml/ns/jakartaee/web-app_6_0.xsd"
         version="6.0">
    <servlet>
        <servlet-name>HelloServlet</servlet-name>
        <servlet-class>com.example.HelloServlet</servlet-class>
    </servlet>
    <servlet-mapping>
```

```
        <servlet-name>HelloServlet</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>
</web-app>
```

5. **Сборка и деплой**:

- Собрать: `mvn clean package` (создает my-web-app-1.0-SNAPSHOT.war в target/).
- Деплой: Скопировать WAR в tomcat/webapps/.
- Доступ: http://localhost:8080/my-web-app/hello.

Как это работает

- Tomcat запускает контейнер сервлетов, загружает WAR.
- При запросе /hello Tomcat вызывает doGet() или doPost().
- Servlet обрабатывает параметры, формирует ответ (HTML, JSON).
- Жизненный цикл: init() при старте, service() для запросов, destroy() при остановке.

Дополнительные возможности

- **Filters**: Перехват запросов/ответов (логирование, авторизация).
- **Listeners**: Реакция на события (запуск/остановка приложения).
- **Sessions**: HttpSession для хранения состояния.
- **JSP**: Для динамических страниц (MVC).
- **REST с JAX-RS**: Интеграция с Tomcat (добавить jersey-servlet).

Пример Filter:

```
import jakarta.servlet.*;
import jakarta.servlet.annotation.WebFilter;
import java.io.IOException;

@WebFilter("/hello")
public class LogFilter implements Filter {
    @Override
    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
chain)
            throws IOException, ServletException {
        System.out.println("Request received at " + System.currentTimeMillis());
        chain.doFilter(req, resp);  // Продолжить
    }
}
```

Tomcat конфигурация

- **server.xml**: Настройка портов, коннекторов (HTTP/HTTPS).
- **context.xml**: Ресурсы (JDBC DataSource).
- **webapps/**: Где деплоятся WAR.

Плюсы Servlets/Tomcat: Простота, масштабируемость, стандарты Jakarta EE. Минусы: Низкоуровневые, для сложных приложений лучше Spring. Ссылки: Tomcat — https://tomcat.apache.org/tomcat-10.1-doc/index.html; Oracle Servlet — https://docs.oracle.com/en/java/javase/21/docs/api/jakarta.servlet/jakarta/servlet/package-summary.html.

Задание Коллекции и StreamAPI

1) Реализация собственного аналога HashSet и ArrayList

Я реализую простые версии коллекций, чтобы продемонстрировать внутреннюю механику. Это не полноценные аналоги из java.util (без generics для простоты, но можно добавить), а образовательные примеры. Я выбрал MyHashSet на основе хэширования с chaining (массив списков для коллизий) и MyArrayList на основе динамического массива. Для MyHashSet: методы add (вставить, без дубликатов) и remove. Для MyArrayList: add, get, remove, addAll.

Реализации основаны на стандартных алгоритмах: для HashSet — хэш-таблица (O(1) average для операций), для ArrayList — ресайзинг массива (amortized O(1) для add). Я объясню код шагово, почему так (для понимания concurrency-safe в будущем, но здесь базово). Ссылки: Baeldung на custom HashSet — https://www.baeldung.com/java-hashset-internal-implementation; Oracle на ArrayList — https://docs.oracle.com/javase/21/docs/api/java/util/ArrayList.html.

MyHashSet

- **Структура**: Массив бакетов (Node[]), каждый — linked list для коллизий.
- **add**: Вычисляет хэш, проверяет наличие, добавляет если нет (избегает дубликатов via equals/hashCode).
- **remove**: Находит и удаляет.
- **Почему так**: Хэширование обеспечивает быстроту; chaining решает коллизии. Без resize для простоты (в реальном HashSet есть load factor 0.75 для resize).

Пример кода:

```
class Node {
    Object value;
    Node next;
    Node(Object value) { this.value = value; }
}

class MyHashSet {
```

```
    private static final int BUCKETS = 16;  // Начальный размер
    private Node[] table = new Node[BUCKETS];
    private int size = 0;

    private int getBucketIndex(Object obj) {
        int hash = (obj == null) ? 0 : obj.hashCode();
        return (hash & (BUCKETS - 1));  // Маска для индекса
    }

    public boolean add(Object obj) {
        int index = getBucketIndex(obj);
        Node node = table[index];
        while (node != null) {
            if ((obj == null && node.value == null) || (obj != null &&
obj.equals(node.value))) {
                return false;  // Уже есть
            }
            node = node.next;
        }
        Node newNode = new Node(obj);
        newNode.next = table[index];
        table[index] = newNode;
        size++;
        return true;
    }

    public boolean remove(Object obj) {
        int index = getBucketIndex(obj);
        Node prev = null;
        Node node = table[index];
        while (node != null) {
            if ((obj == null && node.value == null) || (obj != null &&
obj.equals(node.value))) {
                if (prev == null) {
                    table[index] = node.next;
                } else {
                    prev.next = node.next;
                }
                size--;
                return true;
            }
            prev = node;
            node = node.next;
```

```
        }
        return false;
    }

    public int size() { return size; }
}

// Использование
public class MyHashSetDemo {
    public static void main(String[] args) {
        MyHashSet set = new MyHashSet();
        set.add("apple");
        set.add("banana");
        set.add("apple");  // Не добавит
        System.out.println(set.size());  // 2
        set.remove("banana");
        System.out.println(set.size());  // 1
    }
}
```

MyArrayList

- **Структура**: Массив Object[], capacity удваивается при заполнении.
- **add**: Добавляет в конец, ресайз если нужно.
- **get**: По индексу (O(1)).
- **remove**: По индексу, сдвигает элементы (O(n)).
- **addAll**: Добавляет коллекцию (здесь MyArrayList для примера).
- **Почему так**: Динамический массив для быстрого доступа; ресайз (grow) амортизирует стоимость. Remove сдвигает для сохранения порядка.

Пример кода:

```
import java.util.Arrays;

class MyArrayList {
    private Object[] array;
    private int size = 0;
    private static final int INITIAL_CAPACITY = 10;

    public MyArrayList() {
        array = new Object[INITIAL_CAPACITY];
    }
```

```
    private void grow() {
        int newCapacity = array.length * 2;
        array = Arrays.copyOf(array, newCapacity);
    }

    public void add(Object obj) {
        if (size == array.length) {
            grow();
        }
        array[size++] = obj;
    }

    public Object get(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }
        return array[index];
    }

    public void remove(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }
        System.arraycopy(array, index + 1, array, index, size - index - 1);
        array[--size] = null;  // GC
    }

    public void addAll(MyArrayList other) {
        for (int i = 0; i < other.size; i++) {
            add(other.get(i));
        }
    }

    public int size() { return size; }
}

// Использование
public class MyArrayListDemo {
    public static void main(String[] args) {
        MyArrayList list = new MyArrayList();
        list.add("apple");
        list.add("banana");
        System.out.println(list.get(0));  // apple
```

```
        list.remove(0);
        System.out.println(list.get(0));  // banana

        MyArrayList other = new MyArrayList();
        other.add("cherry");
        other.add("date");
        list.addAll(other);
        System.out.println(list.size());  // 3 (banana, cherry, date)
    }
}
```

2) Класс Student и обработка с одним стримом

Сначала определим классы. Book: с названием, страницами, годом. Переопределим equals/hashCode по названию (для unique), toString. Implements Comparable по страницам (для sorted, но используем Comparator для гибкости).

Student: имя, List. toString для вывода.

Заполним List с 3 студентами, каждый с 5+ книгами (некоторые дубликаты для distinct).

Затем один стрим: students.stream()

- .peek(System.out::println) — вывод студентов (peek — промежуточный, не меняет стрим, для side-effect). Почему peek: позволяет логировать без переменных, следует functional style.
- .map(Student::getBooks) — списки книг.
- .flatMap(List::stream) — flatten в Stream.
- .sorted(Comparator.comparingInt(Book::getPages)) — сортировка по страницам (comparingInt для int). Почему Comparator: гибче Comparable, не меняет класс.
- .distinct() — уникальные (требует equals/hashCode).
- .filter(b -> b.getYear() > 2000) — после 2000.
- .limit(3) — первые 3.
- Теперь "Получить из книг годы выпуска" — но дальше Optional от книги. Чтобы совместить, после limit используем .findFirst() (short-circuiting: останавливает стрим рано, как onlyIfNeeded). findFirst возвращает Optional.
- Почему findFirst: short-circuit, эффективно для больших данных (не обрабатывает весь стрим). Альтернативы: findAny (параллельный), но findFirst ordered.
- Затем для Optional: ifPresentOrElse для вывода года или сообщения. Или get() с orElseThrow, но здесь orElse.

Почему такая реализация: Один стрим для цепочки операций (функциональный стиль Java 8+), минимизирует мутабельность. Peek для side-effect без break chain. Short-circuiting (findFirst) —

оптимизация, останавливает при первом матче. Кто за что: Stream API — абстракция обработки; Comparator — сравнение; Optional — безопасное handling null.

Ссылки: Oracle Stream — https://docs.oracle.com/javase/tutorial/collections/streams/index.html; Baeldung short-circuit — https://www.baeldung.com/java-stream-operations-short-circuit; Optional — https://docs.oracle.com/javase/21/docs/api/java/util/Optional.html (ifPresentOrElse с Java 9).

Пример кода:

```
import java.util.*;
import java.util.stream.Collectors;

class Book {
    private String title;
    private int pages;
    private int year;

    public Book(String title, int pages, int year) {
        this.title = title;
        this.pages = pages;
        this.year = year;
    }

    public int getPages() { return pages; }
    public int getYear() { return year; }
    public String getTitle() { return title; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Book book = (Book) o;
        return Objects.equals(title, book.title);  // Unique по названию
    }

    @Override
    public int hashCode() {
        return Objects.hash(title);
    }

    @Override
    public String toString() {
        return "Book{" + "title='" + title + "', pages=" + pages + ", year=" + year
+ '}';
```

```
    }
}

class Student {
    private String name;
    private List<Book> books;

    public Student(String name, List<Book> books) {
        this.name = name;
        this.books = books;
    }

    public List<Book> getBooks() { return books; }

    @Override
    public String toString() {
        return "Student{" + "name='" + name + "', books=" + books + '}';
    }
}

public class StreamDemo {
    public static void main(String[] args) {
        // Заполнение
        List<Student> students = Arrays.asList(
            new Student("Alice", Arrays.asList(
                new Book("Book1", 100, 1999),
                new Book("Book2", 200, 2001),
                new Book("Book3", 150, 2005),
                new Book("Book4", 300, 1995),
                new Book("Book5", 250, 2010)
            )),
            new Student("Bob", Arrays.asList(
                new Book("Book2", 200, 2001),  // Дубликат
                new Book("Book6", 400, 2003),
                new Book("Book7", 120, 1998),
                new Book("Book8", 180, 2015),
                new Book("Book9", 220, 2002)
            )),
            new Student("Charlie", Arrays.asList(
                new Book("Book3", 150, 2005),  // Дубликат
                new Book("Book10", 350, 2000),
                new Book("Book11", 280, 2018),
                new Book("Book12", 190, 1997),
```

```
                new Book("Book13", 260, 2004)
            ))
        );

        // Один стрим
        students.stream()
                .peek(System.out::println)  // Вывод студентов
                .map(Student::getBooks)     // Списки книг
                .flatMap(List::stream)      // Flatten книг
                .sorted(Comparator.comparingInt(Book::getPages))  // Сорт по
страницам
                .distinct()                 // Unique
                .filter(b -> b.getYear() > 2000)  // После 2000
                .limit(3)                   // 3 элемента
                .findFirst()                // Short-circuit: Optional<Book>
(первый год, но от книги)
                .ifPresentOrElse(
                    book -> System.out.println("Found book year: " +
book.getYear()),  // Вывод года
                    () -> System.out.println("No such book found after processing")
                );
    }
}
```

Объяснение: Стрим обрабатывает цепочкой. findFirst() возвращает Optional от первой Book после limit (годы implicitly через book.getYear()). Если пусто — сообщение. Нет промежуточных vars (всё в chain). Для "Получить из книг годы выпуска" — это implicitly в Optional, фокус на годе. Если нужно list годов — заменить findFirst на collect(Collectors.toList()), но задание на Optional от книги.

Задание try with resources & exceptions

Ваша реализация программы для записи и чтения данных из файла с использованием собственного исключения в целом правильная, но есть несколько моментов, которые можно улучшить для большей надежности, читаемости и соответствия лучшим практикам Java. Я начну с анализа вашего кода, предложу улучшения, а затем предоставлю полный исправленный и оптимизированный вариант с подробными пояснениями. После этого я реализую задание заново с учетом всех требований, чтобы показать альтернативный подход, и объясню, почему выбраны такие решения. Все будет в стиле предыдущих ответов: с наглядными примерами, ссылками на источники и акцентом на принципы ООП.

Анализ вашего кода

Положительные стороны:

1. **CustomException**: Вы создали собственное исключение, унаследованное от Exception (checked exception), что соответствует заданию. Добавление customExceptionCode — хорошая идея для категоризации ошибок.
2. **Использование Files API**: Вы используете современный java.nio.file.Files для операций с файлами, что предпочтительнее старого java.io.File.
3. **Обработка исключений**: Вы ловите IOException и оборачиваете его в CustomException, что правильно для задания.
4. **Try-catch**: Вы обрабатываете FileAlreadyExistsException отдельно, что демонстрирует понимание специфики ошибок.

Проблемы и предложения по улучшению:

1. **Проблема с путем к файлу**:
   - Абсолютный путь `/Volumes/DEV/Java/Projects/FileWriter/textFile.txt` хардкодится и специфичен для вашей системы (macOS). Это не переносимо. Лучше использовать относительный путь или системные свойства (System.getProperty("user.dir")).
   - Нет проверки существования директории перед созданием файла. Если директория не существует, Files.createFile выбросит NoSuchFileException.
2. **CustomException**:
   - Конструктор с кодом ошибки не используется в полной мере (код 102 повторяется для разных ошибок). Лучше сделать уникальные коды или enum для типов ошибок.
   - Сообщения в исключениях начинаются с маленькой буквы ("couldn't"), что нарушает конвенции (должно быть "Couldn't"). См. Oracle Code Conventions — https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html.
   - Нет причины для checked исключения в main (throws CustomException). Можно обрабатывать внутри try-catch для graceful завершения.
3. **Логика обработки ошибок**:
   - FileAlreadyExistsException обрабатывается только выводом в консоль, но программа продолжает выполнение. Лучше либо пропустить создание, либо уведомить пользователя и завершить.
   - Одинаковый код ошибки (102) для чтения и записи затрудняет отладку. Разные операции — разные коды.
   - Нет finally или try-with-resources для Files.write, хотя это не критично здесь (Files.write атомарна).
4. **Читаемость и стиль**:
   - Комментарии вроде "write in fie" содержат опечатки ("file"). Используйте профессиональный стиль.
   - "parcedText" — опечатка ("parsed"). Названия переменных должны быть ясными.
   - Нет проверки на null для textForFile, хотя это маловероятно в данном случае.

- Вывод "file already existing" + path пропущен пробел, что ухудшает читаемость.

5. **Производительность и надежность**:
   - Files.createFile создает файл, но не проверяет права доступа. Можно добавить Files.isWritable().
   - Files.write с StandardOpenOption.WRITE не создаст файл, если он не существует. Нужен StandardOpenOption.CREATE.
6. **ООП и SOLID**:
   - Код в main смешал всю логику (нарушение Single Responsibility Principle). Лучше вынести операции с файлами в отдельный класс (FileHandler).
   - Нет инкапсуляции: логика работы с файлами открыта в main.

Предлагаемые улучшения:

- Использовать относительный путь или создать директорию (Files.createDirectories).
- Улучшить CustomException: уникальные коды ошибок, возможно enum, и капитализация сообщений.
- Использовать try-with-resources для записи/чтения (хотя Files.write/readString не требуют, для единообразия).
- Вынести логику в класс FileHandler (SRP, инкапсуляция).
- Добавить логирование (System.out или Logger) для отладки.
- Убрать throws CustomException из main, обработать внутри.

Исправленный ваш код

Вот исправленная версия вашего кода с учетом замечаний:

```
package ru.swiftvibe;

import java.io.IOException;
import java.nio.file.*;

class CustomException extends Exception {
    private final int errorCode;

    public CustomException(String message, int errorCode) {
        super(message);
        this.errorCode = errorCode;
    }

    public CustomException(String message, int errorCode, Throwable cause) {
        super(message, cause);
        this.errorCode = errorCode;
```

```
    }

    public int getErrorCode() {
        return errorCode;
    }
}

class FileHandler {
    private final Path filePath;

    public FileHandler(String filePath) {
        this.filePath = Paths.get(filePath);
    }

    public void createFile() throws CustomException {
        try {
            Files.createDirectories(filePath.getParent()); // Создать директорию
            Files.createFile(filePath);
            System.out.println("File created successfully at " +
filePath.toAbsolutePath());
        } catch (FileAlreadyExistsException e) {
            System.out.println("File already exists at " +
filePath.toAbsolutePath());
        } catch (IOException e) {
            throw new CustomException("Couldn't create file: " + e.getMessage(),
101, e);
        }
    }

    public void writeToFile(String content) throws CustomException {
        try {
            Files.write(filePath, content.getBytes(), StandardOpenOption.CREATE,
StandardOpenOption.TRUNCATE_EXISTING);
            System.out.println("Successfully wrote to file");
        } catch (IOException e) {
            throw new CustomException("Couldn't write to file: " + e.getMessage(),
102, e);
        }
    }

    public String readFromFile() throws CustomException {
        try {
            return Files.readString(filePath);
```

```
        } catch (IOException e) {
            throw new CustomException("Couldn't read from file: " + e.getMessage(),
103, e);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        String filePath = "textFile.txt"; // Относительный путь
        String textForFile = "This is text that will be written to file and read
from it";
        FileHandler fileHandler = new FileHandler(filePath);

        try {
            fileHandler.createFile();
            fileHandler.writeToFile(textForFile);
            String parsedText = fileHandler.readFromFile();
            System.out.println("TEXT PARSED: " + parsedText);
        } catch (CustomException e) {
            System.err.println("Error [" + e.getErrorCode() + "]: " +
e.getMessage());
            if (e.getCause() != null) {
                e.getCause().printStackTrace();
            }
        }
    }
}
```

Почему так:

- **FileHandler**: Инкапсулирует операции с файлами (SRP). Легко тестировать и переиспользовать.
- **CustomException**: Добавлен конструктор с cause (для chain исключений), уникальные коды (101, 102, 103). Капитализация сообщений.
- **Путь**: Относительный (textFile.txt), создается директория. Переносимо.
- **StandardOpenOption.CREATE**: Позволяет писать в несуществующий файл.
- **Обработка ошибок**: Полная в main, без throws. Выводит код ошибки и stack trace.
- **Логирование**: System.out для успеха, System.err для ошибок.

Новая реализация с учетом задания

Теперь реализую задание заново, чтобы показать альтернативный подход с акцентом на модульность, try-with-resources и современные практики. Программа будет записывать/читать данные, оборачивать ошибки в CustomException, использовать BufferedWriter/BufferedReader (для демонстрации работы с потоками). Реализация включает:

- Создание файла, если не существует.
- Запись списка строк (для гибкости).
- Чтение файла построчно.
- Собственное исключение с enum для кодов ошибок (лучше, чем int).
- Логирование через java.util.logging.Logger.

Enum для ошибок

Используем enum для категоризации ошибок (более типобезопасно, чем int). Это улучшает читаемость и предотвращает дублирование кодов.

FileHandler

Класс инкапсулирует работу с файлами (SRP, инкапсуляция). Используем try-with-resources для потоков. Проверяем права доступа.

CustomException

Checked исключение с enum-кодом и cause для детального stack trace.

Код программы

```
import java.io.*;
import java.nio.file.*;
import java.util.*;
import java.util.logging.Logger;

enum ErrorCode {
    CREATE_FAILED(101, "File creation failed"),
    WRITE_FAILED(102, "File write operation failed"),
    READ_FAILED(103, "File read operation failed");

    private final int code;
    private final String description;

    ErrorCode(int code, String description) {
        this.code = code;
        this.description = description;
    }
```

```
    public int getCode() { return code; }
    public String getDescription() { return description; }
}

class CustomException extends Exception {
    private final ErrorCode errorCode;

    public CustomException(ErrorCode errorCode, String message, Throwable cause) {
        super(message, cause);
        this.errorCode = errorCode;
    }

    public ErrorCode getErrorCode() { return errorCode; }
}

class FileHandler {
    private static final Logger LOGGER =
Logger.getLogger(FileHandler.class.getName());
    private final Path filePath;

    public FileHandler(String filePath) {
        this.filePath = Paths.get(filePath);
    }

    public void createFile() throws CustomException {
        try {
            Files.createDirectories(filePath.getParent());
            if (!Files.exists(filePath)) {
                Files.createFile(filePath);
                LOGGER.info("File created at " + filePath.toAbsolutePath());
            } else {
                LOGGER.info("File already exists at " + filePath.toAbsolutePath());
            }
        } catch (IOException e) {
            throw new CustomException(ErrorCode.CREATE_FAILED,
                "Couldn't create file: " + e.getMessage(), e);
        }
    }

    public void writeToFile(List<String> lines) throws CustomException {
        if (!Files.isWritable(filePath)) {
            throw new CustomException(ErrorCode.WRITE_FAILED,
                "File is not writable: " + filePath, null);
```

```
        }
        try (BufferedWriter writer = Files.newBufferedWriter(filePath,
                StandardOpenOption.CREATE, StandardOpenOption.TRUNCATE_EXISTING)) {
            for (String line : lines) {
                writer.write(line);
                writer.newLine();
            }
            LOGGER.info("Successfully wrote " + lines.size() + " lines to file");
        } catch (IOException e) {
            throw new CustomException(ErrorCode.WRITE_FAILED,
                "Couldn't write to file: " + e.getMessage(), e);
        }
    }

    public List<String> readFromFile() throws CustomException {
        if (!Files.isReadable(filePath)) {
            throw new CustomException(ErrorCode.READ_FAILED,
                "File is not readable: " + filePath, null);
        }
        try (BufferedReader reader = Files.newBufferedReader(filePath)) {
            List<String> lines = new ArrayList<>();
            String line;
            while ((line = reader.readLine()) != null) {
                lines.add(line);
            }
            LOGGER.info("Successfully read " + lines.size() + " lines from file");
            return lines;
        } catch (IOException e) {
            throw new CustomException(ErrorCode.READ_FAILED,
                "Couldn't read from file: " + e.getMessage(), e);
        }
    }
}

public class FileIODemo {
    private static final Logger LOGGER =
Logger.getLogger(FileIODemo.class.getName());

    public static void main(String[] args) {
        FileHandler fileHandler = new FileHandler("data/output.txt");
        List<String> data = Arrays.asList(
            "Line 1: Hello, World!",
            "Line 2: Java I/O Example",
```

```
            "Line 3: Testing file operations"
        );

        try {
            fileHandler.createFile();
            fileHandler.writeToFile(data);
            List<String> readData = fileHandler.readFromFile();
            LOGGER.info("Read content:");
            readData.forEach(line -> LOGGER.info(line));
        } catch (CustomException e) {
            LOGGER.severe(String.format("Error [%s - %d]: %s",
                e.getErrorCode().getDescription(),
                e.getErrorCode().getCode(),
                e.getMessage()));
            if (e.getCause() != null) {
                e.getCause().printStackTrace();
            }
        }
    }
}

<xaiArtifact artifact_id="29e230f9-44e4-4553-9f02-137c6d22a9b0"
artifact_version_id="b183cfcb-8c27-405d-bb97-2be01c713017" title="FileIODemo.java"
contentType="text/java">
import java.io.*;
import java.nio.file.*;
import java.util.*;
import java.util.logging.Logger;

enum ErrorCode {
    CREATE_FAILED(101, "File creation failed"),
    WRITE_FAILED(102, "File write operation failed"),
    READ_FAILED(103, "File read operation failed");

    private final int code;
    private final String description;

    ErrorCode(int code, String description) {
        this.code = code;
        this.description = description;
    }

    public int getCode() { return code; }
```

```
    public String getDescription() { return description; }
}

class CustomException extends Exception {
    private final ErrorCode errorCode;

    public CustomException(ErrorCode errorCode, String message, Throwable cause) {
        super(message, cause);
        this.errorCode = errorCode;
    }

    public ErrorCode getErrorCode() { return errorCode; }
}

class FileHandler {
    private static final Logger LOGGER =
Logger.getLogger(FileHandler.class.getName());
    private final Path filePath;

    public FileHandler(String filePath) {
        this.filePath = Paths.get(filePath);
    }

    public void createFile() throws CustomException {
        try {
            Files.createDirectories(filePath.getParent());
            if (!Files.exists(filePath)) {
                Files.createFile(filePath);
                LOGGER.info("File created at " + filePath.toAbsolutePath());
            } else {
                LOGGER.info("File already exists at " + filePath.toAbsolutePath());
            }
        } catch (IOException e) {
            throw new CustomException(ErrorCode.CREATE_FAILED,
                "Couldn't create file: " + e.getMessage(), e);
        }
    }

    public void writeToFile(List<String> lines) throws CustomException {
        if (!Files.isWritable(filePath)) {
            throw new CustomException(ErrorCode.WRITE_FAILED,
                "File is not writable: " + filePath, null);
        }
```

```
        try (BufferedWriter writer = Files.newBufferedWriter(filePath,
                StandardOpenOption.CREATE, StandardOpenOption.TRUNCATE_EXISTING)) {
            for (String line : lines) {
                writer.write(line);
                writer.newLine();
            }
            LOGGER.info("Successfully wrote " + lines.size() + " lines to file");
        } catch (IOException e) {
            throw new CustomException(ErrorCode.WRITE_FAILED,
                "Couldn't write to file: " + e.getMessage(), e);
        }
    }

    public List<String> readFromFile() throws CustomException {
        if (!Files.isReadable(filePath)) {
            throw new CustomException(ErrorCode.READ_FAILED,
                "File is not readable: " + filePath, null);
        }
        try (BufferedReader reader = Files.newBufferedReader(filePath)) {
            List<String> lines = new ArrayList<>();
            String line;
            while ((line = reader.readLine()) != null) {
                lines.add(line);
            }
            LOGGER.info("Successfully read " + lines.size() + " lines from file");
            return lines;
        } catch (IOException e) {
            throw new CustomException(ErrorCode.READ_FAILED,
                "Couldn't read from file: " + e.getMessage(), e);
        }
    }
}

public class FileIODemo {
    private static final Logger LOGGER =
Logger.getLogger(FileIODemo.class.getName());

    public static void main(String[] args) {
        FileHandler fileHandler = new FileHandler("data/output.txt");
        List<String> data = Arrays.asList(
            "Line 1: Hello, World!",
            "Line 2: Java I/O Example",
            "Line 3: Testing file operations"
```

```
        );

        try {
            fileHandler.createFile();
            fileHandler.writeToFile(data);
            List<String> readData = fileHandler.readFromFile();
            LOGGER.info("Read content:");
            readData.forEach(line -> LOGGER.info(line));
        } catch (CustomException e) {
            LOGGER.severe(String.format("Error [%s - %d]: %s",
                e.getErrorCode().getDescription(),
                e.getErrorCode().getCode(),
                e.getMessage()));
            if (e.getCause() != null) {
                e.getCause().printStackTrace();
            }
        }
    }
}
</xaiArtifact>
```

Почему такая реализация:

- **FileHandler**: Инкапсуляция (SRP из SOLID). Легко расширять (например, добавить сериализацию). Полиморфизм: можно создать подклассы для других форматов.
- **CustomException с enum**: Типобезопасно, описательно. ErrorCode улучшает отладку (101–103 для разных операций). cause сохраняет оригинальную ошибку (IOException).
- **try-with-resources**: Гарантирует закрытие потоков (правило I/O). BufferedWriter/Reader для построчной работы, более явной демонстрации потоков (чем Files.readString).
- **Проверка прав**: Files.isWritable/isReadable предотвращает ошибки до операции.
- **Логирование**: Logger вместо System.out — стандарт для production (настраиваемость). См. Oracle Logging — https://docs.oracle.com/javase/21/docs/api/java/util/logging/Logger.html.
- **Относительный путь**: data/output.txt создается в проекте, переносимо. Files.createDirectories обеспечивает существование папки.
- **List < String >**: Гибкость для записи/чтения (не одна строка). Поддерживает Stream API для обработки.
- **Обработка ошибок**: Все исключения в main, вывод с кодом и описанием. Stack trace для отладки.

Кто за что отвечает:

- **FileHandler**: Инкапсуляция операций I/O, проверка прав, создание директорий, логирование успехов.

- **CustomException**: Стандартизированная обработка ошибок, сохранение cause для детальной диагностики.
- **ErrorCode**: Категоризация ошибок, улучшение читаемости и отладки.
- **Logger**: Логирование операций и ошибок, отделение от бизнес-логики (SRP).
- **main**: Координация, вызов методов, обработка ошибок.

Ссылки:

- Oracle Files API: https://docs.oracle.com/javase/21/docs/api/java/nio/file/Files.html
- Baeldung I/O: https://www.baeldung.com/java-file-nio
- Oracle Exceptions: https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html
- Baeldung Custom Exceptions: https://www.baeldung.com/java-new-custom-exception

Дополнительно:

- Ваш код использовал Files.readString/write, что проще, но менее демонстрирует потоки. Мой вариант с BufferedWriter/Reader показывает низкоуровневую работу, но можно комбинировать (например, Files.write для записи, BufferedReader для чтения).
- Для production лучше добавить валидацию входных данных (null-check для lines).
- Если нужно, можно добавить методы для append (StandardOpenOption.APPEND) или работы с другими форматами (JSON, CSV).

Задание с LiveLock и DeadLock

1) Реализация программ с DeadLock и LiveLock

Deadlock и Livelock — проблемы многопоточности, упомянутые ранее. Здесь я реализую простые программы для демонстрации каждой. Код написан для иллюстрации, и я объясню, почему они приводят к проблемам. Для тестирования я могу мысленно симулировать, но в реальности запустите в IDE (как IntelliJ) и используйте инструменты вроде jstack для анализа deadlock. Livelock может потребовать наблюдения CPU usage (потоки активны, но бесполезны).

Эти примеры используют принципы ООП: классы для задач, но фокусируются на concurrency. Ссылки: Baeldung Deadlock — https://www.baeldung.com/java-deadlock-livelock-starvation; Oracle Concurrency — https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html.

Программа с DeadLock

Deadlock возникает, когда два+ потока ждут друг друга (циклическая зависимость от ресурсов). Здесь два потока захватывают два lock в обратном порядке.

Пример кода:

```
class DeadlockDemo {
    private final Object lock1 = new Object();
    private final Object lock2 = new Object();

    public void methodA() {
        synchronized (lock1) {
            System.out.println(Thread.currentThread().getName() + " acquired
lock1");
            try { Thread.sleep(100); } catch (InterruptedException e) {}  //
Задержка для симуляции
            synchronized (lock2) {
                System.out.println(Thread.currentThread().getName() + " acquired
lock2");
            }
        }
    }

    public void methodB() {
        synchronized (lock2) {
            System.out.println(Thread.currentThread().getName() + " acquired
lock2");
            try { Thread.sleep(100); } catch (InterruptedException e) {}
            synchronized (lock1) {
                System.out.println(Thread.currentThread().getName() + " acquired
lock1");
            }
        }
    }
}

public class DeadlockExample {
    public static void main(String[] args) {
        DeadlockDemo demo = new DeadlockDemo();

        Thread t1 = new Thread(() -> demo.methodA(), "Thread-1");
        Thread t2 = new Thread(() -> demo.methodB(), "Thread-2");

        t1.start();
        t2.start();
    }
}
```

Объяснение:

- Thread-1 захватывает lock1, затем ждет lock2.
- Thread-2 захватывает lock2, затем ждет lock1.
- Результат: Deadlock — потоки заблокированы навсегда. Вывод остановится после первых захватов.
- Почему: Нарушение условия "hold and wait" (удержание и ожидание). Избегайте: Захватывайте locks в одном порядке, используйте tryLock().

Запустите: Программа зависнет. Для анализа: `jps` для PID, `jstack PID` покажет deadlock.

Программа с LiveLock

Livelock: Потоки активны (не blocked), но не прогрессируют, постоянно меняя состояние (например, "вежливо" уступая).

Пример: Два потока пытаются получить доступ, но если другой активен, уступают и пробуют снова.

Пример кода:

```
class LivelockDemo {
    private boolean resource1Available = false;
    private boolean resource2Available = false;

    public void thread1Action() {
        while (true) {
            System.out.println("Thread-1 trying to acquire resource1");
            if (!resource1Available) {
                resource1Available = true;
                if (resource2Available) {
                    System.out.println("Thread-1 acquired both!");
                    break;
                } else {
                    System.out.println("Thread-1 yielding resource1");
                    resource1Available = false;  // Уступает
                    try { Thread.sleep(50); } catch (InterruptedException e) {}
                }
            }
        }
    }

    public void thread2Action() {
        while (true) {
            System.out.println("Thread-2 trying to acquire resource2");
            if (!resource2Available) {
```

```
                resource2Available = true;
                if (resource1Available) {
                    System.out.println("Thread-2 acquired both!");
                    break;
                } else {
                    System.out.println("Thread-2 yielding resource2");
                    resource2Available = false;  // Уступает
                    try { Thread.sleep(50); } catch (InterruptedException e) {}
                }
            }
        }
    }
}

public class LivelockExample {
    public static void main(String[] args) {
        LivelockDemo demo = new LivelockDemo();

        Thread t1 = new Thread(demo::thread1Action, "Thread-1");
        Thread t2 = new Thread(demo::thread2Action, "Thread-2");

        t1.start();
        t2.start();
    }
}
```

Объяснение:

- Каждый поток пытается захватить свой ресурс, но если видит, что другой ресурс захвачен, уступает свой.
- Результат: Livelock — бесконечный цикл "trying/yielding", CPU загружен, но прогресса нет.
- Почему: "Busy waiting" с взаимными уступками. Избегайте: Используйте рандомные задержки или приоритеты.

Запустите: Увидите бесконечный вывод "trying/yielding", но без завершения.

2) Два потока, выводящие "1" и "2" по очереди бесконечно

Задача: Два потока — один печатает "1", другой "2" — чередуясь бесконечно, начиная с "1" (1,2,1,2,...).

Это требует синхронизации для чередования. Я покажу реализацию с потенциалом для livelock/deadlock (плохой пример), затем правильную без проблем.

Плохой пример (с риском Livelock/Deadlock)

Здесь используем флаги и busy-waiting, что может привести к livelock (потоки крутятся в циклах, если timing плохой).

Пример кода:

```
class BadPrinter {
    private volatile boolean isOneTurn = true;  // Начать с 1

    public void printOne() {
        while (true) {
            while (!isOneTurn) {}  // Busy wait
            System.out.print("1");
            isOneTurn = false;
        }
    }

    public void printTwo() {
        while (true) {
            while (isOneTurn) {}  // Busy wait
            System.out.print("2");
            isOneTurn = true;
        }
    }
}

public class BadAlternatingPrint {
    public static void main(String[] args) {
        BadPrinter printer = new BadPrinter();

        Thread t1 = new Thread(printer::printOne, "Thread-1");
        Thread t2 = new Thread(printer::printTwo, "Thread-2");

        t1.start();
        t2.start();
    }
}
```

Объяснение:

- Volatile для видимости.
- Busy-wait (while (!isOneTurn)) — тратит CPU, может привести к livelock если потоки синхронизированы плохо (редко, но возможно в high-load).

- Риск deadlock: Нет, но неэффективно. Может работать, но не рекомендуется.

Правильный пример (без Livelock/Deadlock)

Используем ReentrantLock с двумя Condition для сигнализации. Это обеспечивает правильную очередность без busy-wait (экономит CPU), избегает deadlock (один lock) и livelock (сигналы прогрессируют).

Пример кода:

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class GoodPrinter {
    private final Lock lock = new ReentrantLock();
    private final Condition oneCondition = lock.newCondition();
    private final Condition twoCondition = lock.newCondition();
    private boolean isOneTurn = true;

    public void printOne() {
        lock.lock();
        try {
            while (true) {
                while (!isOneTurn) {
                    oneCondition.await();  // Ждать сигнала
                }
                System.out.print("1");
                isOneTurn = false;
                twoCondition.signal();  // Сигналить второму
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            lock.unlock();
        }
    }

    public void printTwo() {
        lock.lock();
        try {
            while (true) {
                while (isOneTurn) {
                    twoCondition.await();  // Ждать сигнала
```

```
                }
                System.out.print("2");
                isOneTurn = true;
                oneCondition.signal();  // Сигналить первому
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            lock.unlock();
        }
    }
}

public class GoodAlternatingPrint {
    public static void main(String[] args) {
        GoodPrinter printer = new GoodPrinter();

        Thread t1 = new Thread(printer::printOne, "Thread-1");
        Thread t2 = new Thread(printer::printTwo, "Thread-2");

        t1.start();
        t2.start();
    }
}
```

Объяснение:

- Один Lock для mutual exclusion.
- Conditions для ожидания/сигнализации (как wait/notify, но safer).
- Нет busy-wait: await() освобождает lock, спит до signal().
- Избегание deadlock: Один lock, последовательный захват.
- Избегание livelock: Сигналы гарантируют прогресс.
- Вывод: Бесконечно "121212..." без проблем.

Почему правильно: Следует JMM, использует java.util.concurrent для high-level concurrency. Альтернативы: Semaphore (два, по 1 permit), но Condition гибче. Ссылки: Baeldung Condition — https://www.baeldung.com/java-reentrant-lock; Oracle Guarded Blocks — https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html.

Запустите: Увидите чередующийся вывод. Для остановки — Ctrl+C или добавьте флаг завершения.

Задание Паттерны Стратегия, Цепочка обязанностей, Билдер, Прокси, Декоратор, Адаптер

Паттерны проектирования (фокус на указанных)

Паттерны проектирования — это проверенные решения для повторяющихся проблем в ООП. Они помогают соблюдать принципы SOLID и делать код гибким, масштабируемым и поддерживаемым. Указанные паттерны охватывают порождающие (Builder), структурные (Proxy, Decorator, Adapter) и поведенческие (Strategy, Chain of Responsibility). Для каждого я приведу:

- **Определение и назначение**: Что это, когда использовать.
- **Плюсы/минусы**: Преимущества и недостатки.
- **Связь с ООП/SOLID**: Как паттерн использует принципы.
- **Пример реализации**: Полный код на Java с демонстрацией (реальный сценарий, не абстрактный).
- **Объяснение кода**: Шаговый разбор.
- **Ссылки**: Достоверные источники (проверены на актуальность на 2025 год через web_search).

Я использовал поиск для подтверждения ссылок и актуальных примеров.

Стратегия (Strategy)

Определение и назначение: Strategy — поведенческий паттерн, который позволяет определять семейство алгоритмов, инкапсулировать каждый из них и делать их взаимозаменяемыми. Стратегия позволяет алгоритму варьироваться независимо от клиентов, использующих его. Используйте, когда нужно переключать поведение на runtime (например, разные способы сортировки или оплаты). Это реализует принцип Open/Closed (расширение без модификации) и Dependency Inversion (зависимость от абстракций).

Плюсы/минусы:

- Плюсы: Гибкость (легко добавлять новые стратегии), изоляция алгоритмов, упрощает тестирование.
- Минусы: Увеличивает количество классов, клиенты должны знать о стратегиях.

Связь с ООП/SOLID: Полиморфизм (разные реализации одного интерфейса), OCP и DIP из SOLID.

Пример реализации: Сценарий — система сортировки массивов (BubbleSort vs QuickSort). Интерфейс Strategy определяет метод sort(), конкретные классы реализуют алгоритмы, Context использует стратегию.

Полный код:

```java
// Интерфейс стратегии
interface SortStrategy {
    void sort(int[] array);
}

// Конкретная стратегия 1: Bubble Sort
class BubbleSortStrategy implements SortStrategy {
    @Override
    public void sort(int[] array) {
        int n = array.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (array[j] > array[j + 1]) {
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
        System.out.println("Sorted using Bubble Sort");
    }
}

// Конкретная стратегия 2: Quick Sort
class QuickSortStrategy implements SortStrategy {
    private int partition(int[] array, int low, int high) {
        int pivot = array[high];
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (array[j] < pivot) {
                i++;
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
        int temp = array[i + 1];
        array[i + 1] = array[high];
        array[high] = temp;
        return i + 1;
    }

    private void quickSort(int[] array, int low, int high) {
```

```
        if (low < high) {
            int pi = partition(array, low, high);
            quickSort(array, low, pi - 1);
            quickSort(array, pi + 1, high);
        }
    }

    @Override
    public void sort(int[] array) {
        quickSort(array, 0, array.length - 1);
        System.out.println("Sorted using Quick Sort");
    }
}

// Контекст
class Sorter {
    private SortStrategy strategy;

    public void setStrategy(SortStrategy strategy) {
        this.strategy = strategy;
    }

    public void performSort(int[] array) {
        if (strategy != null) {
            strategy.sort(array);
        } else {
            System.out.println("No strategy set");
        }
    }
}

// Демонстрация
public class StrategyDemo {
    public static void main(String[] args) {
        int[] array1 = {5, 3, 8, 4, 2};
        int[] array2 = array1.clone();  // Копия для второго сортировщика

        Sorter sorter = new Sorter();

        // Используем Bubble Sort
        sorter.setStrategy(new BubbleSortStrategy());
        sorter.performSort(array1);
        System.out.println("Result: " + java.util.Arrays.toString(array1));
```

```
        // Переключаем на Quick Sort
        sorter.setStrategy(new QuickSortStrategy());
        sorter.performSort(array2);
        System.out.println("Result: " + java.util.Arrays.toString(array2));
    }
}
```

Объяснение кода:

1. **Интерфейс SortStrategy**: Абстрагирует алгоритм сортировки (метод sort()).
2. **BubbleSortStrategy и QuickSortStrategy**: Конкретные реализации ($O(n^2)$ и $O(n \log n)$ соответственно). Каждая печатает сообщение для демонстрации.
3. **Sorter (Context)**: Хранит ссылку на стратегию, позволяет менять её (setStrategy) и выполнять (performSort). Это обеспечивает динамическое переключение.
4. **Demo**: Создаёт контекст, устанавливает стратегии, сортирует массивы. Вывод: Сначала Bubble, затем Quick на копии массива.

Это демонстрирует, как стратегия меняется без изменения Sorter. В реальности используйте для платежей (CreditCardStrategy vs PayPalStrategy).

Ссылки: Refactoring.Guru (пример Strategy в Java) — https://refactoring.guru/design-patterns/strategy/java/example ; Baeldung (Strategy Pattern) — https://www.baeldung.com/java-strategy-pattern; DigitalOcean (Java Design Patterns) — https://www.digitalocean.com/community/tutorials/java-design-patterns-example-tutorial .

Цепочка обязанностей (Chain of Responsibility)

Определение и назначение: Поведенческий паттерн, который позволяет передавать запросы по цепочке обработчиков. Каждый обработчик решает, обработать запрос или передать дальше. Используйте для обработки событий (логгинг, аутентификация), где порядок важен, но не фиксирован.

Плюсы/минусы:

- Плюсы: Loose coupling (обработчики не знают друг о друге), легко добавлять/удалять.
- Минусы: Запрос может не обработаться, отладка сложнее.

Связь с ООП/SOLID: Полиморфизм (chain через super), SRP (каждый handler — одна ответственность).

Пример реализации: Сценарий — цепочка логгеров (Error -> Warn -> Info). Абстрактный класс Handler, конкретные для уровней.

Полный код:

```java
// Абстрактный обработчик
abstract class LoggerHandler {
    protected LoggerHandler nextHandler;
    protected int level;

    public void setNextHandler(LoggerHandler nextHandler) {
        this.nextHandler = nextHandler;
    }

    public void logMessage(int level, String message) {
        if (this.level <= level) {
            write(message);
        }
        if (nextHandler != null) {
            nextHandler.logMessage(level, message);
        }
    }

    protected abstract void write(String message);
}

// Константы уровней
class LogLevel {
    public static final int INFO = 1;
    public static final int WARN = 2;
    public static final int ERROR = 3;
}

// Конкретный handler: Info
class InfoLogger extends LoggerHandler {
    public InfoLogger(int level) {
        this.level = level;
    }

    @Override
    protected void write(String message) {
        System.out.println("INFO: " + message);
    }
}

// Конкретный handler: Warn
class WarnLogger extends LoggerHandler {
    public WarnLogger(int level) {
```

```
        this.level = level;
    }

    @Override
    protected void write(String message) {
        System.out.println("WARN: " + message);
    }
}

// Конкретный handler: Error
class ErrorLogger extends LoggerHandler {
    public ErrorLogger(int level) {
        this.level = level;
    }

    @Override
    protected void write(String message) {
        System.out.println("ERROR: " + message);
    }
}

// Демонстрация
public class ChainDemo {
    public static void main(String[] args) {
        // Строим цепочку: Error -> Warn -> Info
        LoggerHandler errorLogger = new ErrorLogger(LogLevel.ERROR);
        LoggerHandler warnLogger = new WarnLogger(LogLevel.WARN);
        LoggerHandler infoLogger = new InfoLogger(LogLevel.INFO);

        errorLogger.setNextHandler(warnLogger);
        warnLogger.setNextHandler(infoLogger);

        // Тестируем
        errorLogger.logMessage(LogLevel.INFO, "This is info");
        System.out.println("---");
        errorLogger.logMessage(LogLevel.WARN, "This is warning");
        System.out.println("---");
        errorLogger.logMessage(LogLevel.ERROR, "This is error");
    }
}
```

Объяснение кода:

1. **LoggerHandler**: Базовый класс с nextHandler (для цепочки), logMessage (проверяет уровень, пишет если подходит, передает дальше).
2. **LogLevel**: Константы для уровней (избежание magic numbers).
3. **Конкретные логгеры**: Каждый пишет в своем стиле (INFO/WARN/ERROR).
4. **Chain в main**: Строим цепочку (setNextHandler), стартуем с errorLogger. Запросы проходят по цепи: INFO печатают все, ERROR — только error и ниже.
5. **Вывод**: Для INFO — все три; для ERROR — только ERROR.

Это показывает, как запрос "проходит" цепь. В реальности — для middleware в веб (фильтры).

Ссылки: Refactoring.Guru (Chain of Responsibility) — https://refactoring.guru/design-patterns/chain-of-responsibility ; GeeksforGeeks (Java Chain) — https://www.geeksforgeeks.org/chain-responsibility-design-pattern/; Javatpoint — https://www.javatpoint.com/chain-of-responsibility-pattern .

Билдер (Builder)

Определение и назначение: Порождающий паттерн, который позволяет создавать сложные объекты шаг за шагом. Отделяет конструкцию от представления. Используйте для объектов с множеством параметров (например, конфигурация, immutable объекты), избегая телескопических конструкторов.

Плюсы/минусы:

- Плюсы: Читаемый код, immutable объекты, опциональные параметры.
- Минусы: Больше кода (builder-класс).

Связь с ООП/SOLID: Абстракция (builder интерфейс), OCP (добавление шагов).

Пример реализации: Сценарий — создание Pizza с опциями (size, toppings, cheese). Builder с fluent interface (chain методов).

Полный код:

```
// Продукт
class Pizza {
    private final String size;
    private final String toppings;
    private final boolean extraCheese;
    private final boolean delivery;

    private Pizza(PizzaBuilder builder) {
        this.size = builder.size;
        this.toppings = builder.toppings;
        this.extraCheese = builder.extraCheese;
```

```
        this.delivery = builder.delivery;
    }

    @Override
    public String toString() {
        return "Pizza{size='" + size + "', toppings='" + toppings + "',
extraCheese=" + extraCheese + ", delivery=" + delivery + '}';
    }

    // Builder
    public static class PizzaBuilder {
        private String size = "medium";  // Default
        private String toppings = "";
        private boolean extraCheese = false;
        private boolean delivery = false;

        public PizzaBuilder size(String size) {
            this.size = size;
            return this;
        }

        public PizzaBuilder toppings(String toppings) {
            this.toppings = toppings;
            return this;
        }

        public PizzaBuilder extraCheese(boolean extraCheese) {
            this.extraCheese = extraCheese;
            return this;
        }

        public PizzaBuilder delivery(boolean delivery) {
            this.delivery = delivery;
            return this;
        }

        public Pizza build() {
            return new Pizza(this);
        }
    }
}

// Демонстрация
```

```
public class BuilderDemo {
    public static void main(String[] args) {
        Pizza pizza1 = new Pizza.PizzaBuilder()
                .size("large")
                .toppings("pepperoni, mushrooms")
                .extraCheese(true)
                .build();
        System.out.println(pizza1);

        Pizza pizza2 = new Pizza.PizzaBuilder()
                .delivery(true)
                .build();  // Только default + delivery
        System.out.println(pizza2);
    }
}
```

Объяснение кода:

1. **Pizza**: Immutable (final поля, приватный конструктор). toString для вывода.
2. **PizzaBuilder**: Внутренний статический класс с полями (defaults), fluent методы (return this для chain), build() создаёт Pizza.
3. **Demo**: Строит пиццы шагово. pizza1 — полная, pizza2 — минимальная.
4. **Почему fluent**: Читаемость (как предложение). Defaults позволяют опускать параметры.

В реальности — для StringBuilder или HTTP requests (OkHttp).

Ссылки: Refactoring.Guru (Builder в Java) — https://refactoring.guru/design-patterns/builder/java/example; Baeldung (Builder) — https://www.baeldung.com/creational-design-patterns#builder; DigitalOcean — https://www.digitalocean.com/community/tutorials/java-design-patterns-example-tutorial .

Прокси (Proxy)

Определение и назначение: Структурный паттерн, предоставляющий суррогат или placeholder для другого объекта. Контролирует доступ (lazy load, logging, security). Используйте для remote (RMI), virtual (lazy images), protection (access control).

Плюсы/минусы:

- Плюсы: Добавляет функциональность без изменения оригинала, lazy init.
- Минусы: Overhead, сложность.

Связь с ООП/SOLID: Полиморфизм (один интерфейс), OCP.

Пример реализации: Сценарий — прокси для дорогого ресурса (Image), lazy loading.

Полный код:

```
// Интерфейс
interface Image {
    void display();
}

// Реальный объект
class RealImage implements Image {
    private final String filename;

    public RealImage(String filename) {
        this.filename = filename;
        loadFromDisk();  // Дорогая операция
    }

    private void loadFromDisk() {
        System.out.println("Loading " + filename + " from disk...");
    }

    @Override
    public void display() {
        System.out.println("Displaying " + filename);
    }
}

// Прокси
class ProxyImage implements Image {
    private RealImage realImage;
    private final String filename;

    public ProxyImage(String filename) {
        this.filename = filename;
    }

    @Override
    public void display() {
        if (realImage == null) {
            realImage = new RealImage(filename);  // Lazy load
        }
        realImage.display();
    }
```

```
}

// Демонстрация
public class ProxyDemo {
    public static void main(String[] args) {
        Image image = new ProxyImage("test.jpg");

        // Загрузка только при первом display
        image.display();  // Loading + Displaying
        image.display();  // Только Displaying
    }
}
```

Объяснение кода:

1. **Image**: Общий интерфейс (display()).
2. **RealImage**: Реальный объект, загружает файл в конструкторе (симуляция дорогой операции).
3. **ProxyImage**: Хранит ссылку, создаёт RealImage только при вызове display (lazy). Последующие вызовы — кэшированы.
4. **Demo**: Создаёт прокси, вызывает display дважды. Первый — load + display, второй — только display.

В реальности — для AOP (Spring Proxy) или remote services.

Ссылки: Refactoring.Guru (Proxy в Java) — https://refactoring.guru/design-patterns/proxy/java/example; Baeldung (Proxy) — https://www.baeldung.com/java-proxy-pattern ; GeeksforGeeks — https://www.geeksforgeeks.org/proxy-design-pattern/.

Декоратор (Decorator)

Определение и назначение: Структурный паттерн, позволяющий динамически добавлять новое поведение объектам (wrapper). Используйте для расширения функциональности без подклассов (например, добавление скролла к окну).

Плюсы/минусы:

- Плюсы: Гибкость (комбинации), OCP.
- Минусы: Много мелких классов, сложность трассировки.

Связь с ООП/SOLID: Полиморфизм, OCP, SRP (каждый decorator — одна фича).

Пример реализации: Сценарий — декораторы для кофе (Milk, Sugar добавляют стоимость/описание).

Полный код:

```java
// Компонент
interface Coffee {
    String getDescription();
    double getCost();
}

// Конкретный компонент
class SimpleCoffee implements Coffee {
    @Override
    public String getDescription() {
        return "Simple Coffee";
    }

    @Override
    public double getCost() {
        return 5.0;
    }
}

// Абстрактный декоратор
abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee coffee) {
        this.decoratedCoffee = coffee;
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription();
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost();
    }
}

// Конкретный декоратор: Milk
class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }
```

```
    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Milk";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 2.0;
    }
}

// Конкретный декоратор: Sugar
class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Sugar";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 1.0;
    }
}

// Демонстрация
public class DecoratorDemo {
    public static void main(String[] args) {
        Coffee coffee = new SimpleCoffee();
        System.out.println(coffee.getDescription() + " $" + coffee.getCost());

        // Добавляем декораторы
        coffee = new MilkDecorator(coffee);
        coffee = new SugarDecorator(coffee);
        System.out.println(coffee.getDescription() + " $" + coffee.getCost());
    }
}
```

Объяснение кода:

1. **Coffee**: Интерфейс (getDescription, getCost).
2. **SimpleCoffee**: Базовый объект.
3. **CoffeeDecorator**: Абстрактный wrapper, делегирует decoratedCoffee.
4. **MilkDecorator/SugarDecorator**: Добавляют описание/стоимость.
5. **Demo**: Строит цепь (Simple + Milk + Sugar). Вывод: "Simple Coffee, Milk, Sugar $8.0".

В реальности — для IO (BufferedInputStream декорирует FileInputStream).

Ссылки: Refactoring.Guru (Decorator в Java) — https://refactoring.guru/design-patterns/decorator/java/example; Baeldung (Decorator) — https://www.baeldung.com/java-decorator-pattern ; Medium (2025 примеры) — https://medium.com/codex/10-java-design-patterns-every-senior-engineer-must-master-real-world-examples-a3b4890ce51b .

Адаптер (Adapter)

Определение и назначение: Структурный паттерн, позволяющий объектам с несовместимыми интерфейсами работать вместе. "Переходник". Используйте для интеграции legacy кода или библиотек.

Плюсы/минусы:

- Плюсы: Переиспользование кода, OCP.
- Минусы: Overhead от адаптера.

Связь с ООП/SOLID: Полиморфизм, DIP.

Пример реализации: Сценарий — адаптер для legacy Printer (printString) к новому интерфейсу ModernPrinter (print).

Полный код:

```
// Целевой интерфейс
interface ModernPrinter {
    void print(String message);
}

// Legacy класс
class LegacyPrinter {
    public void printString(String message) {
        System.out.println("Legacy: " + message);
    }
}

// Адаптер (Object Adapter)
```

```
class PrinterAdapter implements ModernPrinter {
    private final LegacyPrinter legacyPrinter;

    public PrinterAdapter(LegacyPrinter legacyPrinter) {
        this.legacyPrinter = legacyPrinter;
    }

    @Override
    public void print(String message) {
        legacyPrinter.printString(message.toUpperCase());  // Адаптация (например, uppercase)
    }
}

// Демонстрация
public class AdapterDemo {
    public static void main(String[] args) {
        LegacyPrinter legacy = new LegacyPrinter();
        ModernPrinter adapter = new PrinterAdapter(legacy);

        adapter.print("Hello, world!");  // Вызов через адаптер
    }
}
```

Объяснение кода:

1. **ModernPrinter**: Новый интерфейс.
2. **LegacyPrinter**: Старый класс с другим методом.
3. **PrinterAdapter**: Реализует ModernPrinter, держит LegacyPrinter, адаптирует вызов (print -> printString + uppercase).
4. **Demo**: Использует адаптер как ModernPrinter.

В реальности — для XML/JSON парсеров.

Ссылки: Refactoring.Guru (Adapter в Java) — https://refactoring.guru/design-patterns/adapter/java/example; Baeldung (Adapter) — https://www.baeldung.com/java-adapter-pattern ; Javatpoint — https://www.javatpoint.com/adapter-pattern.

Понятия

Рефлексия в Java

Определение и назначение: Рефлексия (Reflection) в Java — это механизм, позволяющий программе во время выполнения анализировать и модифицировать собственную структуру:

классы, методы, поля, конструкторы и аннотации. Она предоставляет API (`java.lang.reflect`) для динамической работы с классами, даже если их имена или структура неизвестны на этапе компиляции. Рефлексия используется для:

- Интроспекции (анализ структуры классов).
- Динамического вызова методов/конструкторов.
- Доступа к приватным полям/методам.
- Создания универсальных инструментов (например, DI в Spring, сериализация).

Связь с ООП/SOLID:

- **Полиморфизм**: Динамический вызов методов через рефлексию.
- **Инкапсуляция**: Нарушение (доступ к private), но с осторожностью.
- **DIP**: Рефлексия позволяет зависеть от абстракций (например, интерфейсов).
- **SRP**: Инструменты на рефлексии (например, ORM) разделяют ответственность.

Плюсы/минусы:

- **Плюсы**: Гибкость, поддержка динамических систем (плагины, фреймворки).
- **Минусы**:
  - Производительность ниже (рефлексия медленнее прямых вызовов).
  - Нарушение инкапсуляции (private доступ).
  - Сложность отладки (ошибки на runtime).
  - Безопасность (SecurityManager может ограничивать).

Когда использовать:

- Фреймворки (Spring, Hibernate для DI/ORM).
- Плагины (динамическая загрузка классов).
- Тестирование (доступ к private).
- Сериализация/десериализация (JSON, XML).

Ссылки:

- Oracle Docs (Reflection): https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/reflect/package-summary.html
- Baeldung (Java Reflection): https://www.baeldung.com/java-reflection
- GeeksforGeeks: https://www.geeksforgeeks.org/reflection-in-java/

Основные компоненты рефлексии

1. **Class**: Класс `java.lang.Class` — точка входа. Получаем через:
   - `Class.forName("fully.qualified.ClassName")`
   - `Object.getClass()`
   - `ClassName.class`
2. **Field**: Доступ к полям (включая private).
3. **Method**: Вызов методов (включая private).
4. **Constructor**: Создание объектов через конструкторы.
5. **Annotation**: Анализ аннотаций.

Практические примеры

Я покажу несколько наглядных примеров, демонстрирующих рефлексию:

1. **Интроспекция**: Анализ структуры класса (поля, методы, конструкторы).
2. **Динамический вызов**: Вызов метода по имени.
3. **Доступ к private**: Изменение приватного поля.
4. **Плагины**: Загрузка класса и вызов метода (как в предыдущем примере с ClassLoader).
5. **Аннотации**: Чтение пользовательской аннотации.

Пример 1: Интроспекция класса

Сценарий: Анализируем класс `Person` (поля, методы, конструкторы).

```
import java.lang.reflect.*;

class Person {
    private String name;
    private int age;

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

    private String getName() { return name; }
    public void setAge(int age) { this.age = age; }
    @Override
    public String toString() { return "Person{name='" + name + "', age=" + age +
"}"; }
}
```

```
public class ReflectionIntrospectionDemo {
    public static void main(String[] args) throws ClassNotFoundException {
        // Получаем Class
        Class<?> clazz = Class.forName("Person");

        // Поля
        System.out.println("Fields:");
        for (Field field : clazz.getDeclaredFields()) {
            System.out.println("\t" + Modifier.toString(field.getModifiers()) + " "
+
                               field.getType().getSimpleName() + " " +
field.getName());
        }

        // Конструкторы
        System.out.println("\nConstructors:");
        for (Constructor<?> constructor : clazz.getDeclaredConstructors()) {
            System.out.println("\t" + Modifier.toString(constructor.getModifiers())
+
                               " " + constructor.getName() +
Arrays.toString(constructor.getParameterTypes()));
        }

        // Методы
        System.out.println("\nMethods:");
        for (Method method : clazz.getDeclaredMethods()) {
            System.out.println("\t" + Modifier.toString(method.getModifiers()) + "
" +
                               method.getReturnType().getSimpleName() + " " +
                               method.getName() +
Arrays.toString(method.getParameterTypes()));
        }
    }
}
```

Ожидаемый вывод:

```
Fields:
    private String name
    private int age

Constructors:
    public Person[class java.lang.String, int]
```

```
    public Person[]

Methods:
    private String getName[]
    public void setAge[int]
    public String toString[]
```

Объяснение:

- `Class.forName`: Загружает класс по имени.
- `getDeclaredFields`: Все поля (включая private).
- `getDeclaredConstructors`: Все конструкторы.
- `getDeclaredMethods`: Все методы.
- `Modifier.toString`: Показывает модификаторы (public, private).

Пример 2: Динамический вызов метода

Сценарий: Вызываем метод `toString` и `getName` (private) через рефлексию.

```
import java.lang.reflect.*;

class Person {
    private String name = "Alice";
    private int age = 30;

    private String getName() { return name; }
    @Override
    public String toString() { return "Person{name='" + name + "', age=" + age +
"}"; }
}

public class DynamicMethodInvocationDemo {
    public static void main(String[] args) throws Exception {
        Person person = new Person();
        Class<?> clazz = person.getClass();

        // Вызов public метода toString
        Method toStringMethod = clazz.getMethod("toString");
        String result = (String) toStringMethod.invoke(person);
        System.out.println("toString: " + result);

        // Вызов private метода getName
        Method getNameMethod = clazz.getDeclaredMethod("getName");
        getNameMethod.setAccessible(true); // Обход private
```

```
        String name = (String) getNameMethod.invoke(person);
        System.out.println("getName: " + name);
    }
}
```

Ожидаемый вывод:

```
toString: Person{name='Alice', age=30}
getName: Alice
```

Объяснение:

- `getMethod`: Для public методов (имя + параметры).
- `getDeclaredMethod`: Для любых методов (включая private).
- `setAccessible(true)`: Отключает проверку доступа (может быть запрещено SecurityManager).
- `invoke`: Вызывает метод на объекте, возвращает результат.

Пример 3: Доступ к private полям

Сценарий: Изменяем private поле `name` и читаем `age`.

```
import java.lang.reflect.*;

class Person {
    private String name = "Alice";
    private int age = 30;
}

public class PrivateFieldAccessDemo {
    public static void main(String[] args) throws Exception {
        Person person = new Person();
        Class<?> clazz = person.getClass();

        // Чтение private поля name
        Field nameField = clazz.getDeclaredField("name");
        nameField.setAccessible(true);
        String name = (String) nameField.get(person);
        System.out.println("Original name: " + name);

        // Изменение private поля
        nameField.set(person, "Bob");
        System.out.println("Modified name: " + nameField.get(person));

        // Чтение age
```

```
        Field ageField = clazz.getDeclaredField("age");
        ageField.setAccessible(true);
        int age = (int) ageField.get(person);
        System.out.println("Age: " + age);
    }
}
```

Ожидаемый вывод:

```
Original name: Alice
Modified name: Bob
Age: 30
```

Объяснение:

- `getDeclaredField`: Находит поле по имени.
- `setAccessible(true)`: Доступ к private.
- `get/set`: Чтение/запись значения поля.
- **Осторожно**: Нарушение инкапсуляции может сломать логику класса.

Пример 4: Плагины (динамическая загрузка)

Сценарий: Загружаем класс `PluginImpl` из .class файла (как в примере с ClassLoader) и вызываем метод через рефлексию.

Интерфейс Plugin

```
public interface Plugin {
    void doWork();
}
```

Реализация PluginImpl (отдельный файл, компилируется в plugins/PluginImpl.class)

```
public class PluginImpl implements Plugin {
    @Override
    public void doWork() {
        System.out.println("PluginImpl is doing work!");
    }
}
```

Загрузка и вызов

```
import java.lang.reflect.*;
import java.nio.file.*;

public class PluginReflectionDemo {
```

```
    public static void main(String[] args) throws Exception {
        // Кастомный ClassLoader
        class CustomClassLoader extends ClassLoader {
            private final String classPath;

            public CustomClassLoader(String classPath, ClassLoader parent) {
                super(parent);
                this.classPath = classPath;
            }

            @Override
            protected Class<?> findClass(String name) throws ClassNotFoundException
{
                try {
                    Path filePath = Paths.get(classPath, name.replace('.', '/') +
".class");
                    byte[] classBytes = Files.readAllBytes(filePath);
                    return defineClass(name, classBytes, 0, classBytes.length);
                } catch (IOException e) {
                    throw new ClassNotFoundException("Cannot load class " + name,
e);
                }
            }
        }

        // Загрузка класса
        CustomClassLoader loader = new CustomClassLoader("plugins",
PluginReflectionDemo.class.getClassLoader());
        Class<?> clazz = loader.loadClass("PluginImpl");

        // Проверка интерфейса и вызов метода
        if (Plugin.class.isAssignableFrom(clazz)) {
            Object instance = clazz.getDeclaredConstructor().newInstance();
            Method doWork = clazz.getMethod("doWork");
            doWork.invoke(instance);
        }
    }
}
```

Подготовка:

1. Скомпилируйте `Plugin.java` и `PluginImpl.java`.
2. Переместите `PluginImpl.class` в папку `plugins/`.

3. Запустите `PluginReflectionDemo`.

Ожидаемый вывод:

```
PluginImpl is doing work!
```

Объяснение:

- Кастомный ClassLoader загружает .class файл.
- `getDeclaredConstructor().newInstance()` создает объект.
- `getMethod("doWork").invoke` вызывает метод.
- Полезно для плагинов (например, IntelliJ extensions).

Пример 5: Работа с аннотациями

Сценарий: Создаем аннотацию `@MyAnnotation`, читаем её через рефлексию.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface MyAnnotation {
    String value();
}

class AnnotatedClass {
    @MyAnnotation("important")
    public void importantMethod() {
        System.out.println("Important method called");
    }

    public void normalMethod() {}
}

public class AnnotationReflectionDemo {
    public static void main(String[] args) throws Exception {
        Class<?> clazz = AnnotatedClass.class;

        for (Method method : clazz.getDeclaredMethods()) {
            if (method.isAnnotationPresent(MyAnnotation.class)) {
                MyAnnotation annotation = method.getAnnotation(MyAnnotation.class);
                System.out.println("Method: " + method.getName() + ", Annotation
value: " + annotation.value());
```

```
                method.invoke(new AnnotatedClass());
            }
        }
    }
}
```

Ожидаемый вывод:

```
Method: importantMethod, Annotation value: important
Important method called
```

Объяснение:

- `@Retention(RetentionPolicy.RUNTIME)`: Аннотация доступна на runtime.
- `isAnnotationPresent`: Проверяет наличие аннотации.
- `getAnnotation`: Извлекает аннотацию.
- Используется в Spring (@Autowired), Hibernate (@Entity).

Подводные камни и лучшие практики

Подводные камни

1. **Производительность**: Рефлексия медленнее прямых вызовов (vtable lookup vs reflection overhead). Кэшируйте `Method`, `Field` объекты.
2. **Безопасность**: `setAccessible(true)` может быть запрещено SecurityManager (например, в applets).
3. **Ошибки runtime**: `NoSuchMethodException`, `IllegalAccessException`. Обрабатывайте исключения.
4. **Инкапсуляция**: Доступ к private нарушает дизайн класса, может сломать логику.
5. **ClassLoader**: Разные загрузчики могут вернуть разные Class объекты для одного имени.

Лучшие практики

1. **Кэширование**: Сохраняйте `Class`, `Method`, `Field` в поля для повторного использования.
2. **Проверка типов**: Используйте `isAssignableFrom` для безопасной работы с интерфейсами.
3. **Обработка ошибок**: Всегда обрабатывайте `ReflectiveOperationException` (родитель исключений).
4. **Минимизация private доступа**: Используйте только если нет альтернативы (например, для тестирования).
5. **Аннотации**: Используйте для метаданных вместо хардкода строк.

Пример оптимизации (кэширование)

Сценарий: Кэшируем Method для многократного вызова.

```
import java.lang.reflect.*;

class Optimizer {
    private static final Map<String, Method> methodCache = new HashMap<>();

    public static void invokeCachedMethod(Object obj, String methodName) throws
Exception {
        Method method = methodCache.computeIfAbsent(methodName, name -> {
            try {
                return obj.getClass().getMethod(name);
            } catch (NoSuchMethodException e) {
                throw new RuntimeException(e);
            }
        });
        method.invoke(obj);
    }
}

class Person {
    public void sayHello() {
        System.out.println("Hello!");
    }
}

public class CachedReflectionDemo {
    public static void main(String[] args) throws Exception {
        Person person = new Person();
        for (int i = 0; i < 3; i++) {
            Optimizer.invokeCachedMethod(person, "sayHello");
        }
    }
}
```

Ожидаемый вывод:

```
Hello!
Hello!
Hello!
```

Объяснение: Кэширование `Method` снижает overhead (поиск метода только раз).

Практическое применение

- **Spring/Hibernate**: DI, ORM (чтение аннотаций, вызов сеттеров).
- **Тестирование**: Доступ к private полям/методам (JUnit, TestNG).
- **Плагины**: Динамическая загрузка (IntelliJ, Jenkins).
- **Сериализация**: Jackson/GSON (чтение полей через рефлексию).

Ссылки для углубления:

- Oracle Reflection Guide: https://docs.oracle.com/javase/tutorial/reflect/index.html
- Baeldung (Advanced Reflection): https://www.baeldung.com/java-reflection-performance
- Medium (Reflection Use Cases): https://medium.com/@aarkay0106/java-reflection-practical-applications-4c2f4f8c7a1f