Anubhava Srivastava

# Java 9 Regular Expressions

Zero-length assertions, back-references, quantifiers, and more

Packt

# Регулярные выражения в Java

## Выведение.

Регулярные выражения (или regex) — это мощный инструмент для работы с текстом, который позволяет

По сути, регулярные выражения представляют собой шаблоны, которые описывают набор строк. В языке Java стандартная библиотека предоставляет поддержку для обработки таких выражений через пакет `java.util.regex`.

### Основные компоненты регулярных выражений:

1. **Символьные классы**:
   - `\d` — соответствует любой цифре (0-9).
   - `\D` — соответствует любому символу, который не является цифрой.
   - `\w` — соответствует букве, цифре или символу подчеркивания.
   - `\W` — соответствует любому символу, который не является буквой, цифрой или символом подчеркивания.
   - `\s` — соответствует любому пробельному символу (пробел, табуляция и т. д.).
   - `\S` — соответствует любому непробельному символу.
2. **Квантификаторы**:
   - `*` — соответствует 0 или более экземплярам предыдущего выражения.
   - `+` — соответствует 1 или более экземплярам предыдущего выражения.
   - `?` — соответствует 0 или 1 экземплярам предыдущего выражения.
   - `{n}` — соответствует ровно n экземплярам.
   - `{n,}` — соответствует n или более экземплярам.
   - `{n,m}` — соответствует от n до m экземплярам.
3. **Операторы**:
   - `.` — соответствует любому одиночному символу, кроме перевода строки.
   - `^` — указывает на начало строки.
   - `\$` — указывает на конец строки.
   - `|` — логический “ИЛИ”; позволяет указать альтернативы.

### Применение регулярных выражений в Java

Для работы с регулярными выражениями в Java существует три основных класса:

1. **Pattern** — компилирует регулярное выражение и хранит его в форме, которая может быть использована для поиска.
2. **Matcher** — осуществляет поиск по строке с использованием скомпилированного шаблона.
3. **PatternSyntaxException** — исключение, связанное с ошибками в синтаксисе регулярного выражения.

### Пример использования регулярных выражений

Рассмотрим, как можно использовать регулярные выражения для проверки формата email.

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class EmailValidator {
```

```
    private static final String EMAIL_REGEX = "^[\\w-\\.]+@([\\w-]+\\.)+
[\\w-]{2,4}\$";

    public static boolean isValidEmail(String email) {
        Pattern pattern = Pattern.compile(EMAIL_REGEX);
        Matcher matcher = pattern.matcher(email);
        return matcher.matches();
    }

    public static void main(String[] args) {
        String email = "example@test.com";

        if (isValidEmail(email)) {
            System.out.println(email + " is a valid email address.");
        } else {
            System.out.println(email + " is not a valid email address.");
        }
    }
}
```

В этом коде:

- Мы объявили регулярное выражение для валидации формата почтового адреса.
- Метод `isValidEmail` использует классы `Pattern` и `Matcher` для выполнения проверки.
- В `main` методе мы проверяем, является ли переданный адрес электронной почты допустимым.

### Полезные советы

1. **Отладка**: Регулярные выражения могут быть сложными. Используйте онлайн-валидаторы и инструменты, такие как regex101.com, для отладки ваших шаблонов.

2. **Чтение и запись**: Следуйте принципу «читаемости». Если ваше регулярное выражение становится слишком сложным, разделите его на несколько компонентов и добавьте комментарии.

3. **Оптимизация**: Не создавайте регулярные выражения на лету в циклах, так как это может привести к снижению производительности. Лучше компилировать их один раз и переиспользовать.

### Заключение

Регулярные выражения в Java — это мощный инструмент, который может значительно облегчить обработку строк и текстов. Правильное использование regex позволяет создавать надежные и эффективные приложения для работы с текстовыми данными. Они находят применение в валидации входных данных, парсинге текстов, обработке логов и многом другом. Освоив регулярные выражения, вы сможете значительно расширить свои возможности как разработчика.

(Напишем оглавление 10 глав по 10 подглав в каждой главе)

🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥

Отличная заметка по регулярным выражениям в Java! Она охватывает основные концепции, применение и полезные советы.

Исходя из предоставленной информации и для создания всеобъемлющего руководства, я подготовил оглавление из 10 глав, каждая из которых содержит по 10 подглав:

---

# Оглавление: Регулярные Выражения в Java

## Глава 1: Введение в Регулярные Выражения

1.1. Что такое регулярные выражения (Regex)? 1.2. История и эволюция Regex. 1.3. Основные задачи, решаемые с помощью Regex. 1.4. Преимущества и недостатки использования Regex. 1.5. Регулярные выражения в различных языках программирования. 1.6. Обзор пакета `java.util.regex`. 1.7. Установка и настройка окружения для работы с Java Regex. 1.8. Фундаментальные принципы сопоставления. 1.9. Понятия “шаблон” и “совпадение”. 1.10. Примеры из реального мира: где применяется Regex.

## Глава 2: Базовый Синтаксис Регулярных Выражений

2.1. Символьные литералы и метасимволы. 2.2. Символьные классы: `\d`, `\s`, `\w` и их инверсии. 2.3. Символьные наборы `[]` и диапазоны символов. 2.4. Точка `.` – сопоставление любого символа. 2.5. Квантификаторы: `*`, `+`, `?`. 2.6. Точные квантификаторы: `{n}`, `{n,}`, `{n,m}`. 2.7. Якорные символы: `^` (начало строки) и `\$` (конец строки). 2.8. Альтернатива: оператор `|` (ИЛИ). 2.9. Экранирование специальных символов. 2.10. Понятия жадных, ленивых и сверхагрессивных квантификаторов.

## Глава 3: Продвинутый Синтаксис Регулярных Выражений

3.1. Группировка и захват: `()`. 3.2. Обратные ссылки на захваченные группы. 3.3. Незахватывающие группы: `(?:...)`. 3.4. Опережающие проверки (Lookahead assertions): позитивные и негативные. 3.5. Ретроспективные проверки (Lookbehind assertions): позитивные и негативные. 3.6. Атомарные группы: `(?>...)`. 3.7. Условные выражения в Regex. 3.8. Флаги и режимы сопоставления (`CASE_INSENSITIVE`, `MULTILINE`, `DOTALL` и т.д.). 3.9. Комментарии в регулярных выражениях. 3.10. Использование `\b` для границ слов.

## Глава 4: Класс `Pattern` в Java

4.1. Обзор класса `java.util.regex.Pattern`. 4.2. Метод `compile()`: компиляция регулярного выражения. 4.3. Опции компиляции: их применение и влияние. 4.4. `Pattern.matches()`: простая проверка на полное совпадение. 4.5. `Pattern.split()`: разделение строк по шаблону. 4.6. Получение информации о скомпилированном шаблоне. 4.7. Обработка исключений `PatternSyntaxException`. 4.8. Потокобезопасность `Pattern` и `Matcher`. 4.9. Рекомендации по переиспользованию объектов `Pattern`. 4.10. Создание шаблонов для сложных сценариев.

## Глава 5: Класс `Matcher` в Java

5.1. Обзор класса `java.util.regex.Matcher`. 5.2. Создание объекта `Matcher` из `Pattern`. 5.3. Метод `find()`: поиск следующего совпадения. 5.4. Методы `matches()` и `lookingAt()`: различия и применение. 5.5. Получение найденных групп: `group()`, `group(int)`, `groupCount()`. 5.6. Определение позиций совпадений: `start()`, `end()`. 5.7. Методы замены: `replaceAll()` и `replaceFirst()`. 5.8. Построение замены с помощью `appendReplacement()` и `appendTail()`. 5.9. Метод `reset()` для повторного использования `Matcher`. 5.10. Итерация по всем совпадениям в строке.

## Глава 6: Практические Применения Регулярных Выражений в Java

6.1. Валидация форматов данных: общие подходы. 6.2. Пример: Валидация email-адресов. 6.3. Пример: Валидация телефонных номеров. 6.4. Пример: Валидация URL-адресов и IP-адресов. 6.5. Парсинг и извлечение данных из текстовых файлов (логи, конфиги). 6.6. Поиск и замена текста в строках. 6.7. Разделение строк на токены или компоненты. 6.8. Массовое

переименование файлов. 6.9. Обработка данных из HTML/XML (с оговорками). 6.10. Задачи по обработке естественного языка (NLP): токенизация, выделение сущностей.

## Глава 7: Производительность и Оптимизация Regex в Java

7.1. Влияние регулярных выражений на производительность. 7.2. Проблема “катастрофического отката” (catastrophic backtracking) и как ее избежать. 7.3. Выбор между жадными, ленивыми и сверхагрессивными квантификаторами для оптимизации. 7.4. Использование атомарных групп для предотвращения отката. 7.5. Компиляция шаблонов один раз: `Pattern.compile()`. 7.6. Когда избегать Regex: альтернативы для простых задач. 7.7. Оптимизация сложных регулярных выражений: пошаговый подход. 7.8. Профилирование производительности Regex-кода. 7.9. Бенчмаркинг различных подходов к сопоставлению. 7.10. Советы по написанию эффективных и читаемых Regex-шаблонов.

## Глава 8: Отладка и Тестирование Регулярных Выражений

8.1. Типичные ошибки при написании Regex. 8.2. Использование онлайн-валидаторов и визуализаторов Regex (например, regex101.com). 8.3. Пошаговая отладка: анализ процесса сопоставления. 8.4. Написание юнит-тестов для регулярных выражений. 8.5. Тестирование граничных условий и некорректных входных данных. 8.6. Использование логгирования для отслеживания поведения Regex. 8.7. Разделение сложных Regex на более мелкие, управляемые части. 8.8. Техники для улучшения читаемости Regex: комментарии и форматирование. 8.9. Автоматизированное тестирование Regex-шаблонов. 8.10. Методологии отладки сложных систем с Regex.

## Глава 9: Продвинутые Темы и Особенности Java Regex

9.1. Поддержка Unicode в Java Regex: `\p{L}`, `\p{Nd}` и другие классы. 9.2. Работа с различными кодировками символов. 9.3. Regex и интернационализация (i18n): проблемы и решения. 9.4. Особенности поведения Regex в разных версиях Java. 9.5. Сравнение `java.util.regex` с другими Regex-движками. 9.6. Создание пользовательских символьных классов. 9.7. Использование Regex с потоками данных (Streams API). 9.8. Интеграция Regex с другими Java-библиотеками (например, Apache Commons Lang). 9.9. Динамическое создание регулярных выражений. 9.10. Regex и безопасность: предотвращение Regex Denial of Service (ReDoS) атак.

## Глава 10: Расширенные Сценарии Использования и Лучшие Практики

10.1. Полный пример: Построение простого парсера данных. 10.2. Полный пример: Рефакторинг кода с использованием Regex. 10.3. Регулярные выражения в работе с базами данных (например, PostgreSQL ~). 10.4. Использование Regex в скриптах сборки (Ant, Maven, Gradle). 10.5. Регулярные выражения в IDE (IntelliJ IDEA, Eclipse) для поиска и замены. 10.6. Design Patterns, включающие Regex. 10.7. Сравнение Regex с более мощными инструментами парсинга (ANTLR, JavaCC). 10.8. Инструменты командной строки для работы с Regex (`grep`, `sed`, `awk`). 10.9. Обзор сообщества и ресурсов по Regex. 10.10. Перспективы развития регулярных выражений и их применения.

---

🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥

Отлично! Вот первая глава, посвященная введению в регулярные выражения:

---

# Глава 1: Введение в Регулярные Выражения

## 1.1. Что такое регулярные выражения (Regex)?

Регулярные выражения, часто сокращаемые как Regex или Regexp, представляют собой последовательности символов, которые формируют шаблон поиска. Этот шаблон используется для выполнения операций над текстом, таких как поиск, замена или проверка на соответствие определенному формату. По сути, это мини-язык программирования для описания текстовых паттернов. Их можно сравнить с мощными символами-шаблонами, которые позволяют находить и манипулировать строками гораздо более гибко и эффективно, чем простые текстовые поиски.

## 1.2. История и эволюция Regex

Концепция регулярных выражений берет свое начало в математической логике и теоретической информатике. В 1950-х годах американский математик Стивен Клини формализовал идею регулярных языков и регулярных множеств, которые стали теоретической основой для Regex. Практическое применение регулярные выражения нашли в 1960-х годах в Unix-системах, особенно в таких утилитах, как `grep` (Global Regular Expression Print), `sed` (Stream Editor) и `awk`. С тех пор они были адаптированы и расширены для использования в большинстве современных языков программирования, включая Java, Python, Perl, JavaScript, Ruby и другие, становясь неотъемлемой частью инструментария разработчика.

## 1.3. Основные задачи, решаемые с помощью Regex

Регулярные выражения являются универсальным инструментом для множества задач, связанных с обработкой текста:

- **Валидация данных**: Проверка, соответствует ли введенный пользователем текст определенному формату (например, email, телефонный номер, URL, дата).
- **Поиск и извлечение информации**: Нахождение определенных фрагментов текста в больших объемах данных (например, поиск всех чисел в документе, извлечение заголовков из HTML-страницы).
- **Замена текста**: Массовое изменение текста по заданному шаблону (например, замена всех вхождений “цвет” на “оттенок”).
- **Парсинг данных**: Разделение строки на составные части в соответствии с определенными правилами (например, парсинг лог-файлов или конфигурационных файлов).
- **Рефакторинг кода**: Быстрый поиск и изменение шаблонов в исходном коде программ.

## 1.4. Преимущества и недостатки использования Regex

**Преимущества**:

- **Мощность и гибкость**: Позволяют описывать очень сложные текстовые шаблоны.
- **Краткость**: Часто позволяют выразить сложные логические условия в очень компактной форме.
- **Эффективность**: Реализации Regex-движков обычно высоко оптимизированы для производительности.
- **Переносимость**: Синтаксис Regex стандартизирован и используется во многих языках и инструментах.

**Недостатки**:

- **Сложность**: Сложные регулярные выражения могут быть трудны для чтения, написания и отладки, особенно для новичков.
- **Потенциальные ошибки**: Неправильно написанное Regex может привести к неверным совпадениям или, что хуже, к проблемам с производительностью (например, “катастрофический откат”).
- **Избыточность**: Для простых задач, таких как проверка наличия подстроки, Regex может быть излишним и менее производительным, чем обычные строковые методы.

## 1.5. Регулярные выражения в различных языках программирования

Хотя базовая концепция и синтаксис регулярных выражений остаются в значительной степени одинаковыми, их реализация и доступные возможности могут немного различаться от языка к языку.

- **Perl**: Исторически Perl известен своей мощной встроенной поддержкой Regex, которая оказала большое влияние на другие языки.
- **Python**: Использует модуль `re`, предлагает чистый API.
- **JavaScript**: Встроенная поддержка в виде объекта `RegExp` и методов строк (`match`, `replace`, `split`, `search`).
- **Java**: Предоставляет полноценный пакет `java.util.regex` с классами `Pattern` и `Matcher`.
- **PHP**: Использует функции `preg_*` на основе библиотеки PCRE (Perl Compatible Regular Expressions). Несмотря на различия в API, знание основного синтаксиса Regex позволяет эффективно использовать их в любом из этих языков.

## 1.6. Обзор пакета `java.util.regex`

В Java поддержка регулярных выражений реализована в пакете `java.util.regex`. Этот пакет включает три основных класса:

- **`Pattern`**: Этот класс представляет скомпилированное регулярное выражение. Шаблон должен быть скомпилирован в объект `Pattern` перед использованием для поиска совпадений. Компиляция шаблона повышает производительность, так как JVM не нужно переинтерпретировать шаблон при каждом использовании.
- **`Matcher`**: Этот класс используется для выполнения операций сопоставления шаблона с входной строкой. Объекты `Matcher` создаются из объектов `Pattern` и могут использоваться для поиска, извлечения и замены текста.
- **`PatternSyntaxException`**: Это исключение, которое выбрасывается, если синтаксис регулярного выражения некорректен.

## 1.7. Установка и настройка окружения для работы с Java Regex

Для работы с регулярными выражениями в Java не требуется никакой специальной установки или настройки. Пакет `java.util.regex` является частью стандартной библиотеки Java Development Kit (JDK) и доступен “из коробки” в любом проекте Java. Вам просто нужно импортировать соответствующие классы:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;
// import java.util.regex.PatternSyntaxException; // Обычно явно не
импортируется, так как это исключение
```

Далее вы можете использовать эти классы в своем коде. Например, в любой стандартной IDE (IntelliJ IDEA, Eclipse, NetBeans) или просто компилируя и запуская `.java` файлы из командной строки.

## 1.8. Фундаментальные принципы сопоставления

Процесс сопоставления регулярного выражения с текстом следует определенным принципам:

- **Пошаговое сопоставление**: Regex-движок последовательно просматривает входную строку, пытаясь найти совпадение для шаблона, начиная с первого символа.
- **Жадность по умолчанию**: Квантификаторы (например, `*`, `+`, `{n,m}`) по умолчанию “жадные”, что означает, что они пытаются сопоставить как можно больше символов, при этом обеспечивая, чтобы весь шаблон все еще мог совпасть.
- **Откат (Backtracking)**: Если жадное сопоставление приводит к невозможности найти полное совпадение для всего шаблона, движок “откатывается”, отпуская ранее

захваченные символы, пока не найдет успешное совпадение или не исчерпает все возможности.
- **Соответствие шаблону**: Совпадение считается успешным, если весь шаблон может быть сопоставлен с некоторым фрагментом (или всей) входной строкой.

## 1.9. Понятия “шаблон” и “совпадение”

В контексте регулярных выражений:

- **Шаблон (Pattern)**: Это само регулярное выражение, которое вы определяете. Это последовательность символов, описывающая правила, которым должна соответствовать строка. В Java это объект `Pattern`, созданный методом `Pattern.compile()`.
- **Совпадение (Match)**: Это найденный фрагмент входной строки, который полностью соответствует определенному шаблону. Если шаблон находит несколько таких фрагментов в строке, каждый из них является “совпадением”. В Java объект `Matcher` позволяет вам искать и получать информацию о каждом найденном совпадении.

## 1.10. Примеры из реального мира: где применяется Regex

Регулярные выражения используются повсеместно в различных областях:

- **Веб-разработка**: Валидация форм, парсинг URL-адресов, обработка данных из HTML.
- **Системное администрирование**: Анализ лог-файлов для поиска ошибок или аномалий, фильтрация вывода команд, автоматизация задач.
- **Обработка текстовых данных**: Извлечение информации из больших текстовых корпусов, нормализация данных, поиск дубликатов.
- **Безопасность**: Обнаружение вредоносных шаблонов в сетевом трафике или входных данных (например, SQL-инъекции, XSS-атаки).
- **Разработка программного обеспечения**: Поиск и замена кода в IDE, написание инструментов для генерации кода или анализа.

---

🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥

Отлично! Вот вторая глава, посвященная базовому синтаксису регулярных выражений:

---

# Глава 2: Базовый Синтаксис Регулярных Выражений

## 2.1. Символьные литералы и метасимволы

В основе регулярных выражений лежат символы, которые могут быть двух типов:

- **Символьные литералы (Literal Characters)**: Большинство символов в регулярном выражении соответствуют самим себе. Например, `a`, `B`, `1`, `-`, `_` будут совпадать с соответствующими символами в тексте. Строка `cat` будет совпадать со словом “cat”.
- **Метасимволы (Metacharacters)**: Это специальные символы, которые имеют особое значение в регулярных выражениях и не совпадают сами с собой. Они используются для построения более сложных шаблонов. Примеры метасимволов: `.`, `^`, `\$`, `*`, `+`, `?`, `|`, `(`, `)`, `[`, `]`, `{`, `}`, `\`. Если вы хотите найти сам метасимвол, его необходимо экранировать.

**Пример (Java)**:

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class LiteralExample {
```

```
    public static void main(String[] args) {
        String text = "The quick brown fox jumps over the lazy dog.";

        // Поиск буквальной строки "fox"
        Pattern pattern1 = Pattern.compile("fox");
        Matcher matcher1 = pattern1.matcher(text);
        System.out.println("Contains 'fox': " + matcher1.find()); // true

        // Поиск символа "." (который является метасимволом)
        // Если не экранировать, он будет означать "любой символ"
        Pattern pattern2 = Pattern.compile("\\."); // Экранируем точку
        Matcher matcher2 = pattern2.matcher(text);
        System.out.println("Contains '.': " + matcher2.find()); // true
(найдет точку в конце предложения)
    }
}
```

## 2.2. Символьные классы: `\d`, `\s`, `\w` и их инверсии

Символьные классы (или сокращенные классы символов) предоставляют удобный способ сопоставления с группами символов. Они значительно упрощают написание выражений.

- `\d`: Совпадает с любой **цифрой** (от 0 до 9). Эквивалентно `[0-9]`.
  - `\D`: Совпадает с любым символом, **не являющимся цифрой**. Эквивалентно `[^0-9]`.
- `\s`: Совпадает с любым **пробельным символом** (пробел, табуляция `\t`, перевод строки `\n`, возврат каретки `\r`, символ новой страницы `\f`).
  - `\S`: Совпадает с любым **непробельным символом**.
- `\w`: Совпадает с любой **буквой** (латинский алфавит `a-zA-Z`), **цифрой** (0–9) или **символом подчеркивания** (_). Эквивалентно `[a-zA-Z0-9_]`.
  - `\W`: Совпадает с любым символом, **не являющимся буквой, цифрой или символом подчеркивания**. Эквивалентно `[^a-zA-Z0-9_]`.

**Пример (Java)**:

```
public class CharClassExample {
    public static void main(String[] args) {
        String text = "ID: 12345, Name: John_Doe, Status: Active.";

        // Найти 5 цифр
        Pattern p1 = Pattern.compile("\\d{5}");
        Matcher m1 = p1.matcher(text);
        System.out.println("Found 5 digits: " + m1.find()); // true, найдет
"12345"

        // Найти слово (последовательность \w символов)
        Pattern p2 = Pattern.compile("\\w+"); // + означает один или более
        Matcher m2 = p2.matcher(text);
        while (m2.find()) {
            System.out.println("Word found: " + m2.group()); // ID, 12345,
Name, John_Doe, Status, Active
        }
```

```
    }
}
```

## 2.3. Символьные наборы `[]` и диапазоны символов

Символьные наборы, заключенные в квадратные скобки `[]`, позволяют указать набор символов, любой из которых может быть сопоставлен.

- `[abc]`: Совпадет с `a`, `b` или `c`.
- `[0-9]`: Совпадет с любой цифрой (эквивалентно `\d`).
- `[a-zA-Z]`: Совпадет с любой латинской буквой, как в верхнем, так и в нижнем регистре.
- `[aeiouAEIOU]`: Совпадет с любой гласной.
- `[0-9a-fA-F]`: Совпадет с любой шестнадцатеричной цифрой.

**Отрицание в символьных наборах**: Если первым символом внутри `[]` является `^`, это означает отрицание набора.

- `[^abc]`: Совпадет с любым символом, КРОМЕ `a`, `b` или `c`.
- `[^0-9]`: Совпадет с любым символом, КРОМЕ цифры (эквивалентно `\D`).

**Пример (Java)**:

```
public class CharSetExample {
    public static void main(String[] args) {
        String text = "Color: red, green, Blue, Black.";

        // Найти цвета, начинающиеся с R, G или B (регистронезависимо)
        Pattern p = Pattern.compile("[RrGgBb][a-zA-Z]+"); // Совпадение с R,
G или B, затем одна или более букв
        Matcher m = p.matcher(text);
        while (m.find()) {
            System.out.println("Found color: " + m.group()); // red, green,
Blue, Black
        }

        // Найти все символы, которые не являются буквой или пробелом
        Pattern p2 = Pattern.compile("[^a-zA-Z\\s]");
        Matcher m2 = p2.matcher(text);
        while (m2.find()) {
            System.out.println("Non-alpha/space char: " + m2.group()); // :,
,, ,, .
        }
    }
}
```

## 2.4. Точка `.` – сопоставление любого символа

Метасимвол `.` (точка) является одним из самых универсальных. Он соответствует **любому одиночному символу**, за исключением символа новой строки (`\n`) по умолчанию.

**Пример (Java)**:

```java
public class DotExample {
    public static void main(String[] args) {
        String text1 = "cat";
        String text2 = "cot";
        String text3 = "c@t";
        String text4 = "ct"; // Не совпадет, так как нужен один символ между
c и t

        Pattern p = Pattern.compile("c.t"); // 'c', любой символ, 't'

        System.out.println("cat matches c.t: " +
p.matcher(text1).matches()); // true
        System.out.println("cot matches c.t: " +
p.matcher(text2).matches()); // true
        System.out.println("c@t matches c.t: " +
p.matcher(text3).matches()); // true
        System.out.println("ct matches c.t: " + p.matcher(text4).matches());
// false

        String multiLineText = "Line1\nLine2";
        Pattern p2 = Pattern.compile("Line."); // По умолчанию точка не
совпадает с \n
        Matcher m2 = p2.matcher(multiLineText);
        System.out.println("Line1 matches Line.: " + m2.find()); // true
        System.out.println("Line2 matches Line.: " + m2.find()); // false,
т.к. точка не захватила \n

        // Чтобы точка совпадала со всеми символами, включая \n, используйте
флаг DOTALL (Pattern.DOTALL)
        Pattern p3 = Pattern.compile("Line.", Pattern.DOTALL);
        Matcher m3 = p3.matcher(multiLineText);
        System.out.println("Line1 matches Line. (DOTALL): " + m3.find()); //
true
        System.out.println("Line\n matches Line. (DOTALL): " + m3.find());
// true (захватит "Line\n")
    }
}
```

## 2.5. Квантификаторы: *, +, ?

Квантификаторы определяют, сколько раз предыдущий элемент шаблона должен повторяться.

- * (Звездочка): **Ноль или более** повторений предыдущего элемента.
  - ab*c: ac, abc, abbc, abbbc и т.д.
- + (Плюс): **Один или более** повторений предыдущего элемента.
  - ab+c: abc, abbc, abbbc и т.д. (но не ac).
- ? (Вопросительный знак): **Ноль или одно** повторение предыдущего элемента (делает элемент необязательным).
  - ab?c: ac, abc.

**Пример (Java):**

```
public class QuantifierExample {
    public static void main(String[] args) {
        String text = "abbc ac abc abbbc";

        // ab*c: a, ноль или более b, c
        Pattern p1 = Pattern.compile("ab*c");
        Matcher m1 = p1.matcher(text);
        while (m1.find()) {
            System.out.println("ab*c found: " + m1.group()); // abbc, ac,
abc, abbbc
        }

        // ab+c: a, один или более b, c
        Pattern p2 = Pattern.compile("ab+c");
        Matcher m2 = p2.matcher(text);
        while (m2.find()) {
            System.out.println("ab+c found: " + m2.group()); // abbc, abc,
abbbc (ac не найден)
        }

        // ab?c: a, ноль или одно b, c
        Pattern p3 = Pattern.compile("ab?c");
        Matcher m3 = p3.matcher(text);
        while (m3.find()) {
            System.out.println("ab?c found: " + m3.group()); // abbc
(захватит только ab), ac, abc, abbbc (захватит только ab)
            // Примечание: p3 найдет 'abc' в "abbc" и 'abc' в "abbbc", так
как '?' делает 'b' необязательным
            // и берет только одно 'b' если оно есть, или ноль.
            // При поиске "abbc" шаблон "ab?c" сначала сопоставит "abc",
оставив одно "b" нетронутым.
            // Если вы хотите точное совпадение, используйте `matches()` или
более точный шаблон.
            // Output for ab?c: abbc, ac, abc, abbbc
            // Corrected interpretation for 'ab?c' and 'find()':
            // "abbc": find 'abc' (first 'b' is optional, second 'b' makes
it fail after 'a', then it backs up to 'b')
            // "abbc" -> "ab?c" matches "abc" (the first 'b' is captured as
the 'b?' part, 'c' matches 'c')
            // "ac"   -> "ab?c" matches "ac" ('b?' matches 0 'b's)
            // "abc"  -> "ab?c" matches "abc" ('b?' matches 1 'b')
            // "abbbc" -> "ab?c" matches "abc" (first 'b' captured)
            // The output will be: abbc, ac, abc, abbbc
        }
    }
}
```

## 2.6. Точные квантификаторы: `{n}`, `{n,}`, `{n,m}`

Эти квантификаторы предоставляют более точный контроль над количеством повторений.

- `{n}`: Совпадает ровно с `n` повторениями предыдущего элемента.
  - `a{3}`: aaa
- `{n,}`: Совпадает с `n` или более повторениями предыдущего элемента.
  - `a{2,}`: aa, aaa, aaaa и т.д.
- `{n,m}`: Совпадает от `n` до `m` повторений предыдущего элемента (включительно).
  - `a{2,4}`: aa, aaa, aaaa.

**Пример (Java)**:

```
public class ExactQuantifierExample {
    public static void main(String[] args) {
        String text = "aaa aaaaaa aa a";

        // a{3}: ровно 3 'a'
        Pattern p1 = Pattern.compile("a{3}");
        Matcher m1 = p1.matcher(text);
        while (m1.find()) {
            System.out.println("a{3} found: " + m1.group()); // aaa, aaa (из
aaaaaa)
        }

        // a{2,}: 2 или более 'a'
        Pattern p2 = Pattern.compile("a{2,}");
        Matcher m2 = p2.matcher(text);
        while (m2.find()) {
            System.out.println("a{2,} found: " + m2.group()); // aaa,
aaaaaa, aa
        }

        // a{2,4}: от 2 до 4 'a'
        Pattern p3 = Pattern.compile("a{2,4}");
        Matcher m3 = p3.matcher(text);
        while (m3.find()) {
            System.out.println("a{2,4} found: " + m3.group()); // aaa, aaaa
(из aaaaaa), aa
        }
    }
}
```

## 2.7. Якорные символы: `^` (начало строки) и `\$` (конец строки)

Якорные символы не сопоставляют конкретные символы, а обозначают позиции в строке.

- `^`: Совпадает с **началом строки**.
- `\$`: Совпадает с **концом строки**.

При использовании с методом `Matcher.matches()`, который требует, чтобы весь вход соответствовал шаблону, `^` и `\$` подразумеваются. Однако, при использовании `Matcher.find()`, они явно указывают на начало/конец строки.

**Пример (Java)**:

```
public class AnchorExample {
    public static void main(String[] args) {
        String text1 = "Hello World";
        String text2 = "World is big";

        // ^Hello: строка начинается с "Hello"
        Pattern p1 = Pattern.compile("^Hello");
        Matcher m1 = p1.matcher(text1);
        System.out.println("'Hello World' starts with 'Hello': " +
m1.find()); // true
        Matcher m1_2 = p1.matcher(text2);
        System.out.println("'World is big' starts with 'Hello': " +
m1_2.find()); // false

        // World\$: строка заканчивается на "World"
        Pattern p2 = Pattern.compile("World\$");
        Matcher m2 = p2.matcher(text1);
        System.out.println("'Hello World' ends with 'World': " + m2.find());
// true
        Matcher m2_2 = p2.matcher(text2);
        System.out.println("'World is big' ends with 'World': " +
m2_2.find()); // false

        // ^Hello World\$: вся строка должна быть "Hello World"
        Pattern p3 = Pattern.compile("^Hello World\$");
        Matcher m3 = p3.matcher(text1);
        System.out.println("'Hello World' is exactly 'Hello World': " +
m3.matches()); // true
    }
}
```

## 2.8. Альтернатива: оператор | (ИЛИ)

Оператор `|` действует как логическое “ИЛИ”. Он позволяет указать несколько альтернативных шаблонов, любой из которых может совпасть.

- `cat|dog`: Совпадет со словом “cat” или словом “dog”.
- `gra(y|e)y`: Совпадет с “gray” или “grey”. Скобки `()` используются для группировки альтернатив.

**Пример (Java)**:

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

```
        String text = "The color is red or blue.";

        // Найти "red" или "blue"
        Pattern p = Pattern.compile("red|blue");
        Matcher m = p.matcher(text);
        while (m.find()) {
            System.out.println("Found: " + m.group()); // red, blue
        }

        String browserText = "Using Chrome browser or Firefox browser.";
        // Найти "Chrome" или "Firefox"
        Pattern p2 = Pattern.compile("(Chrome|Firefox) browser"); //
Группировка важна здесь
        Matcher m2 = p2.matcher(browserText);
        while (m2.find()) {
            System.out.println("Browser found: " + m2.group(1)); // Chrome,
Firefox (group(0) would be "Chrome browser", "Firefox browser")
        }
    }
}
```

## 2.9. Экранирование специальных символов

Как упоминалось в разделе 2.1, метасимволы имеют особое значение. Если вам нужно сопоставить сам метасимвол (например, буквальную точку `.`, звездочку `*`, скобку `(` и т.д.), его необходимо **экранировать** с помощью обратной косой черты `\`. В Java, так как обратная косая черта `\` сама является специальным символом в строковых литералах (например, `\n` для новой строки), вам нужно экранировать ее саму. Поэтому для экранирования метасимвола в Regex вы используете **две обратные косые черты** в Java-строке (`\\`).

- Чтобы найти буквальную точку: `\.` (в Java строке: `"\\."`)
- Чтобы найти буквальный плюс: `\+` (в Java строке: `"\\+"`)
- Чтобы найти буквальную открывающую скобку: `LEFTPAREN` (в Java строке: `"\LEFTPAREN"`)

**Пример (Java)**:

```
public class EscapingExample {
    public static void main(String[] args) {
        String text = "Price is \$10.50. Quantity = 5.";

        // Найти "\$10.50"
        // \$: метасимвол (конец строки), .: метасимвол (любой символ)
        Pattern p1 = Pattern.compile("\\$10\\.50"); // Экранируем \$ и .
        Matcher m1 = p1.matcher(text);
        System.out.println("Found price: " + m1.find()); // true
        if (m1.find()) { // move matcher position again
            System.out.println("The found price is: " + m1.group());
        }

        // Найти текст в скобках, например, (value)
```

```
        Pattern p2 = Pattern.compile("\LEFTPARENvalue\RIGHTPAREN"); //
Экранируем ( и )
        Matcher m2 = p2.matcher("This is (value) in parentheses.");
        System.out.println("Found (value): " + m2.find()); // true
    }
}
```

## 2.10. Понятия жадных, ленивых и сверхагрессивных квантификаторов

Квантификаторы по умолчанию в Regex **жадные (greedy)**. Это означает, что они пытаются сопоставить как можно больше текста, при этом позволяя оставшейся части регулярного выражения успешно совпасть.

- **Жадные (Greedy)**: `*`, `+`, `?`, `{n}`, `{n,}`, `{n,m}`. (по умолчанию)

**Пример жадного поведения**: Шаблон `"<.*>"` в строке `"<a> <b>"` Жадный `.*` сначала попытается захватить весь остаток строки: `"<a> <b>"`. Затем `>` не совпадет, и `.*` будет отступать по одному символу, пока не найдет `>` в конце `<b>`. В итоге, он совпадет со всей строкой `"<a> <b>"`.

Чтобы изменить это поведение, можно использовать **ленивые (reluctant/lazy)** квантификаторы, добавляя `?` после жадного квантификатора. Они пытаются сопоставить как можно меньше текста.

- **Ленивые (Reluctant/Lazy)**: `*?`, `+?`, `??`, `{n}?`, `{n,}?`, `{n,m}?`

**Пример ленивого поведения**: Шаблон `"<.*?>"` в строке `"<a> <b>"` Ленивый `.*?` сначала попытается захватить ноль символов, затем один, пока не найдет следующее `>`. Он сначала совпадет с `"<a>"`, затем продолжит поиск и найдет `"<b>"`.

Существуют также **сверхагрессивные (possessive)** квантификаторы (добавляется `+` после жадного). Они похожи на жадные, но после захвата текста они **не позволяют откату (backtracking)**. Если шаблон не может быть сопоставлен после сверхагрессивного квантификатора, то совпадение не будет найдено, даже если откат мог бы помочь. Это повышает производительность в случаях, когда откат нежелателен или не приведет к успеху, но делает их менее гибкими.

- **Сверхагрессивные (Possessive)**: `*+`, `++`, `?+`, `{n}+`, `{n,}+`, `{n,m}+`

**Пример (Java) – Жадные vs. Ленивые**:

```
public class QuantifierTypesExample {
    public static void main(String[] args) {
        String html = "<p>First paragraph</p><p>Second paragraph</p>";

        // Жадный квантификатор: .*
        // Совпадет со всем от первой '<' до последней '>'
        Pattern greedyPattern = Pattern.compile("<.*>");
        Matcher greedyMatcher = greedyPattern.matcher(html);
        if (greedyMatcher.find()) {
            System.out.println("Greedy match: " + greedyMatcher.group());
            // Output: <p>First paragraph</p><p>Second paragraph</p>
        }

        // Ленивый квантификатор: .*?
        // Совпадет с каждым тегом по отдельности
```

```
        Pattern lazyPattern = Pattern.compile("<.*?>");
        Matcher lazyMatcher = lazyPattern.matcher(html);
        while (lazyMatcher.find()) {
            System.out.println("Lazy match: " + lazyMatcher.group());
            // Output: <p>, </p>, <p>, </p>
        }

        // Пример сверхагрессивного квантификатора (для демонстрации
отсутствия отката)
        String textWithNumbers = "123456";
        // Шаблон пытается найти 3+ цифры, за которыми следует 5
        // Жадный: \\d{3,}5 - захватит "1234", затем отступит до "123",
чтобы 5 совпало
        Pattern p1 = Pattern.compile("(\\d{3,})5"); // Группа для
иллюстрации захвата
        Matcher m1 = p1.matcher(textWithNumbers);
        System.out.println("Greedy \\d{3,}5 found: " + m1.find()); // true
        if (m1.find()) { // Reset the matcher to the beginning if re-used or
create new
             m1 = p1.matcher(textWithNumbers);
             m1.find();
            System.out.println("  Greedy match group 1: " + m1.group(1)); //
123
        }

        // Сверхагрессивный: \\d{3,}+5 - захватит "123456", не отступит, 5
не совпадет
        Pattern p2 = Pattern.compile("(\\d{3,}+)5"); // Сверхагрессивный
квантификатор
        Matcher m2 = p2.matcher(textWithNumbers);
        System.out.println("Possessive \\d{3,}+5 found: " + m2.find()); //
false
        // Объяснение: \\d{3,}+ жадно захватывает "123456". Затем шаблон
пытается найти '5'
        // после "123456", но его нет. Так как сверхагрессивный
квантификатор не отступает,
        // совпадение не найдено.
    }
}
```

---

🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥

Отлично! Вот третья глава, посвященная продвинутому синтаксису регулярных выражений:

---

# Глава 3: Продвинутый Синтаксис Регулярных Выражений

## 3.1. Группировка и захват: `()`

Круглые скобки `()` в регулярных выражениях служат двум основным целям:

1. **Группировка (Grouping)**: Они позволяют объединить несколько элементов регулярного выражения в одну логическую единицу. Это полезно, когда вы хотите применить квантификатор к целому выражению или использовать оператор “ИЛИ” `|` для выбора между группами шаблонов.
    - Пример: `(ab)+` совпадет с `ab`, `abab`, `ababab` и т.д. Без скобок `ab+` совпало бы только с `abbb...`.
    - Пример: `(cat|dog)s` совпадет с `cats` или `dogs`.
2. **Захват (Capturing)**: Каждый раз, когда группа совпадает с частью входной строки, эта часть “захватывается” и сохраняется в памяти. Затем к этим захваченным подстрокам можно обращаться по их номеру или имени (если они именованные группы). Группы нумеруются слева направо, начиная с 1, по порядку открывающих скобок. Группа 0 всегда соответствует всему совпадению.

**Пример (Java)**:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class CapturingGroupExample {
    public static void main(String[] args) {
        String text = "Contact us at email@example.com or info@domain.org.";

        // Шаблон для email, захватывающий имя пользователя и домен
        Pattern p = Pattern.compile("(\\w+)@([\\w.-]+\\.\\w{2,4})");
        Matcher m = p.matcher(text);

        while (m.find()) {
            System.out.println("Full match: " + m.group(0)); //
email@example.com
            System.out.println("Username: " + m.group(1));    // email
            System.out.println("Domain: " + m.group(2));      // example.com
            System.out.println("---");
        }
    }
}
```

## 3.2. Обратные ссылки на захваченные группы

Обратные ссылки позволяют ссылаться на ранее захваченную группу внутри того же регулярного выражения. Это полезно для поиска повторяющихся паттернов или дублирующихся слов. Обратные ссылки обозначаются как `\n`, где `n` — это номер захваченной группы.

**Пример (Java)**:

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

```
        String text1 = "apple apple";
        String text2 = "banana orange";
        String text3 = "This is a test test string.";

        // Найти повторяющееся слово: (\w+) захватывает слово, \1 ссылается
на него же
        Pattern p = Pattern.compile("(\\b\\w+)\\s+\\1\\b");
        // \\b - граница слова, \\s+ - один или более пробельных символов

        Matcher m1 = p.matcher(text1);
        System.out.println("Found duplicate in '" + text1 + "': " +
m1.find()); // true

        Matcher m2 = p.matcher(text2);
        System.out.println("Found duplicate in '" + text2 + "': " +
m2.find()); // false

        Matcher m3 = p.matcher(text3);
        System.out.println("Found duplicate in '" + text3 + "': " +
m3.find()); // true
        if (m3.find()) { // Reset the matcher to the beginning if re-used or
create new
            m3 = p.matcher(text3);
            m3.find();
            System.out.println("Duplicate word: " + m3.group(1)); // test
        }
    }
}
```

## 3.3. Незахватывающие группы: `(?:...)`

Иногда вам нужна группировка для применения квантификатора или оператора `|`, но вы не хотите, чтобы эта группа захватывала текст (т.е., чтобы она занимала номер группы и влияла на производительность). Для этого используются незахватывающие группы, обозначаемые как `(?:...)`.

**Пример (Java)**:

```
public class NonCapturingGroupExample {
    public static void main(String[] args) {
        String text = "The quick brown fox jumps over the lazy dog.";

        // (?:quick|brown|lazy) - сгруппировано для 'ИЛИ', но не захватывает
        Pattern p = Pattern.compile("The (?:quick|brown|lazy) (?:fox|dog)");
        Matcher m = p.matcher(text);

        if (m.find()) {
            System.out.println("Full match: " + m.group(0)); // The quick
brown fox
```

```
            System.out.println("Group count: " + m.groupCount()); // 0, так
как нет захватывающих групп
        }

        // Для сравнения, с захватывающими группами:
        Pattern p2 = Pattern.compile("The (quick|brown|lazy) (fox|dog)");
        Matcher m2 = p2.matcher(text);
        if (m2.find()) {
            System.out.println("Full match (capturing): " + m2.group(0));
            System.out.println("Group count (capturing): " +
m2.groupCount()); // 2
            System.out.println("Group 1: " + m2.group(1)); // quick
            System.out.println("Group 2: " + m2.group(2)); // fox
        }
    }
}
```

## 3.4. Опережающие проверки (Lookahead assertions): позитивные и негативные

Lookahead-проверки — это утверждения (assertions), которые проверяют, следует ли за текущей позицией определенный шаблон, но **не включают** этот шаблон в совпадение. Они не потребляют символы.

- **Позитивная опережающая проверка (Positive Lookahead)**: `(?=pattern)`
    - Успех, если `pattern` *следует* за текущей позицией.
    - Пример: `foo(?=bar)` совпадет с `foo` только если за ним следует `bar` (но `bar` не будет частью совпадения).
- **Негативная опережающая проверка (Negative Lookahead)**: `(?!pattern)`
    - Успех, если `pattern` *не следует* за текущей позицией.
    - Пример: `foo(?!bar)` совпадет с `foo` только если за ним *не следует* `bar`.

**Пример (Java)**:

```
public class LookaheadExample {
    public static void main(String[] args) {
        String text = "apple pie, orange juice, banana split";

        // Найти слова, за которыми следует "pie"
        Pattern p1 = Pattern.compile("\\w+(?=\\spie)");
        Matcher m1 = p1.matcher(text);
        while (m1.find()) {
            System.out.println("Word before 'pie': " + m1.group()); // apple
        }

        // Найти слова, за которыми НЕ следует "juice"
        Pattern p2 = Pattern.compile("\\w+\\b(?!\\sjuice)");
        Matcher m2 = p2.matcher(text);
```

```
        while (m2.find()) {
            System.out.println("Word not followed by 'juice': " +
m2.group());
            // apple, pie, orange, banana, split
            // (orange будет найдено, т.к. за ним следует пробел, а не "
juice")
            // Если нужно, чтобы НЕ было слова "juice" после слова:
            // "\\b\\w+\\b(?!\\s+juice)"
        }
    }
}
```

## 3.5. Ретроспективные проверки (Lookbehind assertions): позитивные и негативные

Lookbehind-проверки похожи на lookahead, но они проверяют шаблон, который **предшествует** текущей позиции, не включая его в совпадение. В отличие от lookahead, lookbehind в Java (и многих других реализациях) должен иметь **фиксированную длину** или быть выражением, которое можно определить как фиксированной длины (например, `(?:fixed|length)`).

- **Позитивная ретроспективная проверка (Positive Lookbehind)**: `(?<=pattern)`
    - Успех, если `pattern` *предшествует* текущей позиции.
    - Пример: `(?<=abc)def` совпадет с `def` только если ему предшествует `abc`.
- **Негативная ретроспективная проверка (Negative Lookbehind)**: `(?<!pattern)`
    - Успех, если `pattern` *не предшествует* текущей позиции.
    - Пример: `(?<!abc)def` совпадет с `def` только если ему *не предшествует* `abc`.

**Пример (Java)**:

```
public class LookbehindExample {
    public static void main(String[] args) {
        String text = "USD100, EUR50, JPY200";

        // Найти числа, которым предшествует "USD"
        Pattern p1 = Pattern.compile("(?<=USD)\\d+"); // Найти цифры, если
перед ними "USD"
        Matcher m1 = p1.matcher(text);
        while (m1.find()) {
            System.out.println("USD value: " + m1.group()); // 100
        }

        // Найти числа, которым НЕ предшествует "EUR"
        Pattern p2 = Pattern.compile("(?<!EUR)\\d+");
        Matcher m2 = p2.matcher(text);
        while (m2.find()) {
            System.out.println("Non-EUR value: " + m2.group()); // 100 (для
USD100), 200 (для JPY200)
            // Он также найдет "50" из EUR50, так как (?<!EUR) проверяет
```

```
только 3 символа назад и "50"
            // не имеет "EUR" перед собой (там "UR5", если смотреть 3
символа назад от 50)
            // Если вы хотите, чтобы вся "EUR" не предшествовала: "(?
<!EUR)\\d+" работает, если "EUR" 3 символа.
            // Output would be 100, 50, 200, but the 50 is correct due to
the non-matching part.
            // For a more precise match on whole numbers:
            // Pattern p2_alt = Pattern.compile("(?<!EUR)\\b(\\d+)\\b");
        }
    }
}
```

**Важное замечание о `(?<=...)` и `(?<!...)` в Java**: Содержимое lookbehind assertions в Java должно иметь фиксированную длину. Например, `(?<=\\d+)` (одна или более цифр) вызовет ошибку `PatternSyntaxException`, потому что `\d+` имеет переменную длину. Однако `(?<=\\d{3})` (ровно 3 цифры) будет работать.

## 3.6. Атомарные группы: `(?>...)`

Атомарные группы похожи на незахватывающие группы, но с одним ключевым отличием: они **отключают откат (backtracking)** для шаблона внутри группы. Если часть шаблона внутри атомарной группы совпадает, движок не будет пытаться откатиться, чтобы найти другое совпадение для остальной части регулярного выражения. Это может значительно улучшить производительность в некоторых случаях, когда откат не нужен, или предотвратить “катастрофический откат”.

**Пример (Java)**:

```java
public class AtomicGroupExample {
    public static void main(String[] args) {
        String text = "ababa";

        // Без атомарной группы (будет откат)
        // (?:a|ab)+a - Эта строка будет совпадать с "ababa"
        // (?:a|ab)+: сначала попытается захватить как можно больше.
        // Возьмет 'aba' (a, ba), затем 'a' останется.
        // (a|ab)+a - 'ab' 'ab' 'a'
        Pattern p1 = Pattern.compile("(a|ab)+a");
        Matcher m1 = p1.matcher(text);
        System.out.println("Non-atomic group (a|ab)+a finds 'ababa': " +
m1.matches()); // true

        // С атомарной группой (нет отката)
        // (?>a|ab)+a - захватит как можно больше, но НЕ ОТКАТИТСЯ
        // (?>a|ab)+ попытается совпасть с 'ab' в начале, затем 'ab'
        // Он захватит 'ab' 'ab', останется 'a'.
        // После этого он не сможет найти 'a' в конце шаблона, потому что
        // атомарная группа не вернет символы для отката.
        Pattern p2 = Pattern.compile("(?>a|ab)+a");
        Matcher m2 = p2.matcher(text);
```

```
        System.out.println("Atomic group (?>a|ab)+a finds 'ababa': " +
m2.matches()); // false
        // Объяснение: (?>a|ab)+ жадно захватит "ab" (из 'ababa'), затем
снова "ab" (из 'aba').
        // Останется только "a". Теперь шаблон требует "a" в конце, но все
символы уже захвачены
        // атомарной группой, и она не позволит откатиться. Поэтому
совпадения нет.
    }
}
```

Атомарные группы часто используются в сочетании со сверхагрессивными квантификаторами, поскольку они выполняют схожую функцию, но для целых групп.

## 3.7. Условные выражения в Regex

**Важное замечание для Java**: Стандартный пакет `java.util.regex` **НЕ ПОДДЕРЖИВАЕТ** условные выражения (conditional regex constructs) напрямую. Условные выражения, такие как `(?(condition)true_regex|false_regex)` или `(?(group)true_regex|false_regex)`, являются особенностью других движков регулярных выражений, например, Perl и PCRE (Perl Compatible Regular Expressions), которые используются в PHP и Python.

В Java для реализации условной логики на основе совпадений регулярных выражений вам потребуется использовать программную логику с помощью `if/else` или других управляющих структур Java после получения результатов сопоставления.

**Пример концепции (не в Java Regex, а для понимания):** Представьте, что вы хотите сопоставить номер телефона, который может начинаться с кода страны, но код страны является необязательным. `^(?:(?P<country_code>\+\d{1,3}) )?(?P<area_code>\d{3})-\\d{3}-\\d{4}\$` Если бы условные выражения поддерживались, можно было бы сделать что-то вроде: `^(?(1)\\d{1,3}\\s)?\\d{3}-\\d{3}-\\d{4}\$` (Если группа 1 (country_code) существует, то после нее следует пробел). **В Java такой синтаксис невозможен.**

## 3.8. Флаги и режимы сопоставления

Флаги (или режимы) изменяют поведение регулярного выражения. Их можно установить при компиляции объекта `Pattern`.

- `Pattern.CASE_INSENSITIVE` (или `(?i)`): Игнорирует регистр символов при сопоставлении.
- `Pattern.MULTILINE` (или `(?m)`): Изменяет поведение `^` и `\$` таким образом, что они совпадают с началом/концом каждой строки (после `\n` или `\r`), а не только всей входной строки.
- `Pattern.DOTALL` (или `(?s)`): Изменяет поведение `.` (точки) так, что она совпадает со всеми символами, включая символы новой строки (`\n`).
- `Pattern.UNICODE_CASE` (или `(?u)`): В сочетании с `CASE_INSENSITIVE` включает Unicode-специфичное сопоставление без учета регистра.
- `Pattern.COMMENTS` (или `(?x)`): Включает режим комментариев, который игнорирует пробелы и позволяет использовать комментарии (#) внутри регулярного выражения.
- `Pattern.CANON_EQ`: Включает режим канонической эквивалентности для сопоставления Unicode-символов.

Флаги можно комбинировать с помощью оператора `|`.

**Пример (Java)**:

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

```
        String text1 = "Hello World";
        String text2 = "hello world";
        String multiLineText = "Line1\nLine2\nLine3";

        // CASE_INSENSITIVE
        Pattern p1 = Pattern.compile("hello", Pattern.CASE_INSENSITIVE);
        Matcher m1 = p1.matcher(text1);
        System.out.println("Case insensitive 'hello' in 'Hello World': " +
m1.find()); // true

        // MULTILINE
        // ^Line: начало каждой строки
        Pattern p2 = Pattern.compile("^Line\\d", Pattern.MULTILINE);
        Matcher m2 = p2.matcher(multiLineText);
        while (m2.find()) {
            System.out.println("Multiline match: " + m2.group()); // Line1,
Line2, Line3
        }

        // DOTALL (точка совпадает с \n)
        Pattern p3 = Pattern.compile("Line.\\nLine.", Pattern.DOTALL);
        Matcher m3 = p3.matcher(multiLineText);
        System.out.println("Dotall match (Line.\\nLine.): " + m3.find()); //
true (найдет "Line1\nLine2")
        if (m3.find()) { // Reset matcher
            m3 = p3.matcher(multiLineText);
            m3.find();
            System.out.println("  Match: " + m3.group());
        }
    }
}
```

## 3.9. Комментарии в регулярных выражениях

Для повышения читаемости сложных регулярных выражений можно добавлять комментарии. Это делается с помощью синтаксиса `(?#comment)`. Комментарии игнорируются движком Regex. Комментарии особенно полезны при использовании флага `Pattern.COMMENTS` (`(?x)`), который также позволяет использовать пробелы и символы новой строки для форматирования шаблона и `#` для обычных комментариев (до конца строки).

**Пример (Java)**:

```
public class CommentsExample {
    public static void main(String[] args) {
        String ipAddress = "192.168.1.1";

        // Регулярное выражение для IP-адреса с комментариями
        String ipRegex =
              "^"                     // Начало строки
```

```
            + "([01]?\\d\\d?|2[0-4]\\d|25[0-5])" // Первая октета: 0-255 (?#
First octet)
            + "\\."                    // Разделитель: точка
            + "([01]?\\d\\d?|2[0-4]\\d|25[0-5])" // Вторая октета (?# Second
octet)
            + "\\."
            + "([01]?\\d\\d?|2[0-4]\\d|25[0-5])" // Третья октета (?# Third
octet)
            + "\\."
            + "([01]?\\d\\d?|2[0-4]\\d|25[0-5])" // Четвертая октета (?#
Fourth octet)
            + "\$"                     // Конец строки
            ;

        // Если использовать флаг Pattern.COMMENTS, можно форматировать и
использовать #
        String ipRegexFormatted =
              "^" + // Start of the line
              "([01]?\\d\\d?|2[0-4]\\d|25[0-5])" + // First octet (0-255)
              "\\." + // Dot separator
              "([01]?\\d\\d?|2[0-4]\\d|25[0-5])" + // Second octet
              "\\." +
              "([01]?\\d\\d?|2[0-4]\\d|25[0-5])" + // Third octet
              "\\." +
              "([01]?\\d\\d?|2[0-4]\\d|25[0-5])" + // Fourth octet
              "\$" ; // End of the line

        // В Java комментарии (?#...) и режим Pattern.COMMENTS (x)
        // Для демонстрации (?#comment)
        Pattern p1 = Pattern.compile("apple(?#This is a comment)pie");
        Matcher m1 = p1.matcher("applepie");
        System.out.println("Match with (?#comment): " + m1.matches()); //
true

        // Для демонстрации Pattern.COMMENTS (x)
        // В этом режиме пробелы игнорируются, и # означает комментарий до
конца строки
        Pattern p2 = Pattern.compile(
            "^(?#Start of line)\n" +
            "(\\d{3})" + // First three digits # Example group 1
            "\\s*" +     // Optional space
            "(\\d{3})" + // Second three digits # Example group 2
            "\\s*" +
            "(\\d{4})\$", // Last four digits # Example group 3
            Pattern.COMMENTS
        );
        Matcher m2 = p2.matcher("123 456 7890");
        System.out.println("Match with Pattern.COMMENTS: " + m2.matches());
// true
```

```
    }
}
```

## 3.10. Использование `\b` для границ слов

Метасимвол `\b` представляет собой “границу слова”. Он сопоставляет позицию, где один из символов является символом слова (`\w`), а другой — несимволом слова (`\W`) или началом/концом строки. `\b` полезен для поиска целых слов, а не их частей.

- `\bword\b`: Совпадет с полным словом “word”.
- `\bcat`: Совпадет со словом, которое начинается с “cat” (например, “cat”, “caterpillar”).
- `cat\b`: Совпадет со словом, которое заканчивается на “cat” (например, “cat”, “bobcat”).

**Пример (Java)**:

```
public class WordBoundaryExample {
    public static void main(String[] args) {
        String text = "cat, caterpillar, bobcat, wildcat.";

        // Найти точное слово "cat"
        Pattern p1 = Pattern.compile("\\bcat\\b");
        Matcher m1 = p1.matcher(text);
        System.out.println("Found 'cat' as a whole word: " + m1.find()); //
true (найдет первый "cat")

        // Найти слово, которое начинается с "cat"
        Pattern p2 = Pattern.compile("\\bcat");
        Matcher m2 = p2.matcher(text);
        while (m2.find()) {
            System.out.println("Word starting with 'cat': " + m2.group());
// cat, caterpillar
        }

        // Найти слово, которое заканчивается на "cat"
        Pattern p3 = Pattern.compile("cat\\b");
        Matcher m3 = p3.matcher(text);
        while (m3.find()) {
            System.out.println("Word ending with 'cat': " + m3.group()); //
cat, bobcat, wildcat
        }
    }
}
```

---

🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥

Отлично! Вот четвертая глава, посвященная классу `Pattern` в Java:

---

# Глава 4: Класс `Pattern` в Java

## 4.1. Обзор класса `java.util.regex.Pattern`

Класс `java.util.regex.Pattern` является фундаментом для работы с регулярными выражениями в Java. Его основная роль — представлять собой **скомпилированное регулярное выражение**. Когда вы создаете регулярное выражение в виде строки (например, `"\\d+"`), оно находится в "сыром" текстовом формате. Для эффективного поиска совпадений с этим выражением в тексте, Java Regex-движок должен сначала преобразовать эту текстовую строку в внутреннее, оптимизированное представление, которое он может быстро использовать для выполнения операций сопоставления. Этот процесс называется **компиляцией**.

Объекты `Pattern` являются:

- **Неизменяемыми (Immutable)**: После создания объект `Pattern` не может быть изменен. Это делает его потокобезопасным.
- **Потокобезопасными (Thread-safe)**: Поскольку `Pattern` неизменяем, его можно безопасно использовать несколькими потоками одновременно.

Класс `Pattern` предоставляет статические методы для компиляции и некоторые полезные методы для выполнения простых операций сопоставления и разделения строк.

## 4.2. Метод `compile()`: компиляция регулярного выражения

Метод `compile()` является основным способом создания объекта `Pattern`. Он принимает строковое представление регулярного выражения и компилирует его.

**Синтаксис**:

```
public static Pattern compile(String regex)
public static Pattern compile(String regex, int flags)
```

- `regex`: Строка, содержащая регулярное выражение.
- `flags`: Необязательный аргумент, одно или несколько значений флагов, которые изменяют поведение сопоставления (например, `Pattern.CASE_INSENSITIVE`).

**Пример (Java)**:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class PatternCompileExample {
    public static void main(String[] args) {
        // Компиляция простого регулярного выражения
        Pattern pattern1 = Pattern.compile("abc");
        // Создание Matcher из скомпилированного Pattern
        Matcher matcher1 = pattern1.matcher("abcdef");
        System.out.println("Pattern 'abc' found in 'abcdef': " +
matcher1.find()); // true

        // Компиляция с флагами (например, регистронезависимый поиск)
        Pattern pattern2 = Pattern.compile("hello",
Pattern.CASE_INSENSITIVE);
        Matcher matcher2 = pattern2.matcher("Hello World");
        System.out.println("Pattern 'hello' (case-insensitive) found in
```

```
'Hello World': " + matcher2.find()); // true
    }
}
```

**Важность компиляции**: Компиляция регулярного выражения является ресурсоемкой операцией. Поэтому рекомендуется компилировать шаблон только один раз и переиспользовать полученный объект `Pattern` всякий раз, когда вам нужно выполнить сопоставление с одним и тем же шаблоном.

## 4.3. Опции компиляции: их применение и влияние

Опции компиляции (флаги) позволяют изменять стандартное поведение регулярного выражения. Они передаются в метод `compile()` в виде целочисленных констант. Флаги можно комбинировать с помощью побитового ИЛИ (`|`).

Основные флаги, доступные в классе `Pattern`:

- `Pattern.CASE_INSENSITIVE` (или `(?i)`): Сопоставляет буквы независимо от регистра (A-Z и a-z).
- `Pattern.MULTILINE` (или `(?m)`): Изменяет поведение якорей `^` и `\$` для сопоставления с началом и концом каждой строки, а не только всей входной последовательности.
- `Pattern.DOTALL` (или `(?s)`): Точка `.` совпадает со всеми символами, включая символы новой строки (`\n`). Без этого флага `.` не совпадает с `\n`.
- `Pattern.UNICODE_CASE` (или `(?u)`): В сочетании с `CASE_INSENSITIVE` включает Unicode-специфичное сопоставление без учета регистра.
- `Pattern.COMMENTS` (или `(?x)`): Позволяет игнорировать пробелы и комментарии, начинающиеся с `#` до конца строки, внутри регулярного выражения. Это улучшает читаемость сложных шаблонов.
- `Pattern.UNIX_LINES` (или `(?d)`): Влияет на то, как `.` (если не используется `DOTALL`), `^` и `\$` интерпретируют символы новой строки. Только `\n` рассматривается как терминатор строки.

**Пример (Java)**:

```
public class PatternFlagsExample {
    public static void main(String[] args) {
        String text = "Line one\nline Two\nLINE three";

        // Комбинирование флагов: регистронезависимый и многострочный режим
        Pattern pattern = Pattern.compile("^line", Pattern.CASE_INSENSITIVE
| Pattern.MULTILINE);
        Matcher matcher = pattern.matcher(text);

        while (matcher.find()) {
            System.out.println("Found: " + matcher.group());
            // Output: Line, line, LINE (найдет в начале каждой строки,
независимо от регистра)
        }
    }
}
```

## 4.4. `Pattern.matches()`: простая проверка на полное совпадение

Класс `Pattern` предоставляет удобный статический метод `matches()`, который позволяет быстро проверить, соответствует ли **вся** входная строка заданному регулярному выражению. Этот метод

по сути выполняет компиляцию шаблона, создает `Matcher` и вызывает его метод `matches()` за одну операцию.

**Синтаксис**:

```
public static boolean matches(String regex, CharSequence input)
```

- `regex`: Строка с регулярным выражением.
- `input`: Строка, которую нужно проверить.

**Пример (Java)**:

```
public class PatternMatchesExample {
    public static void main(String[] args) {
        String phoneNumber1 = "123-456-7890";
        String phoneNumber2 = "1234567890";
        String invalidNumber = "abc-def-ghij";

        // Проверка на соответствие формату XYY-YYY-YYYY
        String regex = "\\d{3}-\\d{3}-\\d{4}";

        System.out.println("'" + phoneNumber1 + "' matches: " +
Pattern.matches(regex, phoneNumber1)); // true
        System.out.println("'" + phoneNumber2 + "' matches: " +
Pattern.matches(regex, phoneNumber2)); // false
        System.out.println("'" + invalidNumber + "' matches: " +
Pattern.matches(regex, invalidNumber)); // false

        // Pattern.matches() всегда пытается совпасть со всей строкой
        System.out.println("Pattern.matches(\"\\d+\", \"123\") -> " +
Pattern.matches("\\d+", "123")); // true
        System.out.println("Pattern.matches(\"\\d+\", \"123abc\") -> " +
Pattern.matches("\\d+", "123abc")); // false
        // Эквивалентно Pattern.compile("\\d+").matcher("123abc").matches()
    }
}
```

**Важно**: Метод `matches()` всегда пытается сопоставить **всю** входную строку. Если вам нужно найти подстроку, используйте методы `Matcher.find()` или `Matcher.lookingAt()`.

## 4.5. `Pattern.split()`: разделение строк по шаблону

Метод `split()` в классе `Pattern` используется для разделения входной строки на массив подстрок, используя регулярное выражение в качестве разделителя.

**Синтаксис**:

```
public String[] split(CharSequence input)
public String[] split(CharSequence input, int limit)
```

- `input`: Строка для разделения.
- `limit`: Необязательный аргумент, определяющий максимальное количество частей, на которые будет разделена строка.
    - `limit > 0`: Шаблон будет применен `limit - 1` раз, и массив будет содержать не более `limit` элементов. Последний элемент будет содержать остаток строки.
    - `limit < 0`: Шаблон будет применен столько раз, сколько возможно, и возвращаемый массив может быть любой длины. Пустые строки в конце не будут отброшены.
    - `limit = 0`: То же, что и `limit < 0`, но пустые строки в конце будут отброшены (это поведение по умолчанию, если `limit` не указан).

**Пример (Java)**:

```
public class PatternSplitExample {
    public static void main(String[] args) {
        String data = "apple,banana,,orange,grape";

        // Разделение по запятой
        Pattern p1 = Pattern.compile(",");
        String[] parts1 = p1.split(data);
        System.out.println("Split by ',':");
        for (String s : parts1) {
            System.out.println("  \"" + s + "\""); // "apple", "banana", "",
"orange", "grape"
        }

        System.out.println("\nSplit by ',' with limit 3:");
        String[] parts2 = p1.split(data, 3);
        for (String s : parts2) {
            System.out.println("  \"" + s + "\""); // "apple", "banana",
",,orange,grape"
        }

        System.out.println("\nSplit by ',' (default, discards trailing empty
strings):");
        String dataWithTrailing = "a,b,c,,,";
        String[] parts3 = p1.split(dataWithTrailing); // limit=0 by default
        for (String s : parts3) {
            System.out.println("  \"" + s + "\""); // "a", "b", "c" (пустые
в конце отброшены)
        }

        System.out.println("\nSplit by ',' (limit -1, keeps trailing empty
strings):");
        String[] parts4 = p1.split(dataWithTrailing, -1);
        for (String s : parts4) {
            System.out.println("  \"" + s + "\""); // "a", "b", "c", "", "",
""
        }

        // Разделение по нескольким разделителям (пробел, запятая, точка с
запятой)
```

```
        Pattern p5 = Pattern.compile("[\\s,;]+"); // Один или более
пробелов, запятых или точек с запятой
        String complexData = "word1 word2,word3;word4   word5";
        String[] parts5 = p5.split(complexData);
        System.out.println("\nSplit by '[\\s,;]+':");
        for (String s : parts5) {
            System.out.println("  \"" + s + "\""); // "word1", "word2",
"word3", "word4", "word5"
        }
    }
}
```

## 4.6. Получение информации о скомпилированном шаблоне

Класс `Pattern` также предоставляет методы для получения информации о скомпилированном шаблоне.

- `public String pattern()`: Возвращает строковое представление регулярного выражения, которое было использовано для компиляции этого `Pattern` объекта.
- `public int flags()`: Возвращает целочисленное значение, представляющее флаги, с которыми был скомпилирован `Pattern`. Это значение является комбинацией побитового ИЛИ (`|`) констант флагов.

**Пример (Java)**:

```
public class PatternInfoExample {
    public static void main(String[] args) {
        Pattern pattern = Pattern.compile("test", Pattern.CASE_INSENSITIVE |
Pattern.MULTILINE);

        System.out.println("Original Regex: " + pattern.pattern()); // test
        System.out.println("Flags as int: " + pattern.flags()); // 66
(Pattern.CASE_INSENSITIVE is 2, Pattern.MULTILINE is 8, (2|8) is 10)
        // Hmm, wait, the values are: CASE_INSENSITIVE (2), MULTILINE (8). 2
| 8 = 10.
        // Let's recheck Pattern class source or docs for flag values.
        // Correct values: CASE_INSENSITIVE=2, MULTILINE=8. So 10. My output
'66' was likely a typo or wrong value from memory.
        // It's the sum of the bitmasks. 2 + 8 = 10.
        // The numbers of flags can be arbitrary, usually powers of 2 (1, 2,
4, 8, 16, 32, 64, etc.)
        // CASE_INSENSITIVE = 0x02 (2)
        // MULTILINE = 0x08 (8)
        // DOTALL = 0x20 (32)
        // UNICODE_CASE = 0x04 (4)
        // COMMENTS = 0x04 (4) - no, COMMENTS = 0x08 (8) - wait, this is
important
        // Let's get actual values.
        // Pattern.CASE_INSENSITIVE (2)
        // Pattern.MULTILINE (8)
```

```
        // Pattern.DOTALL (32)
        // Pattern.UNICODE_CASE (4)
        // Pattern.COMMENTS (64)
        // Pattern.UNIX_LINES (1)
        // Pattern.CANON_EQ (128)

        // So, 2 (CASE_INSENSITIVE) | 8 (MULTILINE) = 10. This is the
correct value.
        // My previous thought about "66" was wrong.
        System.out.println("Flags as int: " + pattern.flags()); // Should be
10

        // Проверка конкретного флага
        if ((pattern.flags() & Pattern.CASE_INSENSITIVE) != 0) {
            System.out.println("Pattern is CASE_INSENSITIVE.");
        }
        if ((pattern.flags() & Pattern.MULTILINE) != 0) {
            System.out.println("Pattern is MULTILINE.");
        }
    }
}
```

## 4.7. Обработка исключений `PatternSyntaxException`

`PatternSyntaxException` — это непроверяемое исключение (RuntimeException), которое выбрасывается, когда строка, переданная в метод `Pattern.compile()`, содержит синтаксически некорректное регулярное выражение.

**Пример (Java)**:

```
import java.util.regex.PatternSyntaxException;

public class PatternSyntaxExceptionExample {
    public static void main(String[] args) {
        try {
            // Некорректное регулярное выражение: незакрытая скобка
            Pattern pattern = Pattern.compile("abc(");
            System.out.println("Pattern compiled successfully.");
        } catch (PatternSyntaxException e) {
            System.err.println("Regex syntax error caught!");
            System.err.println("Description: " + e.getDescription()); //
Описание ошибки
            System.err.println("Message: " + e.getMessage());       //
Полное сообщение с позицией
            System.err.println("Index: " + e.getIndex());           //
Позиция ошибки
            System.err.println("Regex: " + e.getPattern());         //
Некорректный шаблон
        }
```

```
        try {
            // Ещё один пример: неверное использование квантификатора
            Pattern pattern = Pattern.compile("*abc"); // Звездочка без
предшествующего элемента
        } catch (PatternSyntaxException e) {
            System.err.println("\nAnother regex syntax error caught!");
            System.err.println("Description: " + e.getDescription());
            System.err.println("Index: " + e.getIndex());
        }
    }
}
```

**Совет**: Всегда старайтесь проверять синтаксис ваших регулярных выражений, особенно при их создании из внешних источников (ввод пользователя, конфигурационные файлы), чтобы избежать `PatternSyntaxException`. Онлайн-валидаторы regex (например, regex101.com) крайне полезны для отладки.

## 4.8. Потокобезопасность `Pattern` и `Matcher`

Понимание потокобезопасности этих двух классов критически важно для многопоточных Java-приложений:

- **`Pattern`**: Объекты `Pattern` **потокобезопасны**. Как упоминалось ранее, они неизменяемы. После того, как вы скомпилировали регулярное выражение в объект `Pattern`, его внутреннее состояние не меняется. Это означает, что один и тот же объект `Pattern` может быть безопасно использован несколькими потоками для создания своих собственных объектов `Matcher`.

- **`Matcher`**: Объекты `Matcher` **НЕ потокобезопасны**. `Matcher` содержит состояние (например, текущую позицию поиска, захваченные группы), которое изменяется при вызове таких методов, как `find()`, `group()`, `reset()`. Если несколько потоков попытаются использовать один и тот же объект `Matcher` одновременно, это приведет к непредсказуемому поведению и ошибкам.

**Рекомендация**:

- Создавайте объекты `Pattern` один раз и храните их (например, в `static final` поле).
- Для каждого потока или для каждой операции сопоставления создавайте новый объект `Matcher` из `Pattern` (с помощью `pattern.matcher(inputString)`).

**Пример (Java)**:

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

public class ThreadSafetyExample {
    // Правильно: Pattern является статическим и неизменяемым
    private static final Pattern EMAIL_PATTERN =
        Pattern.compile("(\\w+)@([\\w.-]+\\.\\w{2,4})");

    public static void main(String[] args) throws InterruptedException {
        String[] emails = {
```

```
            "test1@example.com",
            "user2@domain.org",
            "info@mail.net",
            "invalid-email",
            "another.user@sub.domain.co.uk"
        };

        ExecutorService executor = Executors.newFixedThreadPool(3);

        for (String email : emails) {
            executor.submit(() -> {
                // Правильно: Каждый поток создает свой собственный Matcher
                Matcher matcher = EMAIL_PATTERN.matcher(email);
                if (matcher.matches()) {
                    System.out.println(Thread.currentThread().getName() +
                                       ": '" + email + "' is VALID.
Username: " + matcher.group(1));
                } else {
                    System.out.println(Thread.currentThread().getName() +
                                       ": '" + email + "' is INVALID.");
                }
            });
        }

        executor.shutdown();
        executor.awaitTermination(1, TimeUnit.MINUTES);
    }
}
```

## 4.9. Рекомендации по переиспользованию объектов `Pattern`

Как уже упоминалось, компиляция регулярного выражения — это относительно дорогая операция. Для оптимальной производительности следуйте этим рекомендациям:

1. **Компилируйте один раз**: Если вы будете использовать одно и то же регулярное выражение многократно (например, в цикле, в разных методах или в многопоточном приложении), скомпилируйте его в объект `Pattern` только один раз.

```
// Правильно:
public static final Pattern MY_REGEX = Pattern.compile("somePattern");

public void processText(String text) {
    Matcher matcher = MY_REGEX.matcher(text); // Используем уже
скомпилированный Pattern
    // ...
}
```

```
// Неправильно: Компиляция происходит при каждом вызове, это медленно
public void processTextInefficiently(String text) {
```

```
    Pattern p = Pattern.compile("somePattern");
    Matcher m = p.matcher(text);
    // ...
}
```

2. **Используйте `static final`**: Для выражений, которые используются в классе и не меняются, объявите `Pattern` как `static final` поле. Это гарантирует, что шаблон будет скомпилирован только один раз при загрузке класса.
3. **Избегайте `Pattern.matches()` в циклах для сложных Regex**: Хотя `Pattern.matches()` удобен, он компилирует шаблон при каждом вызове. Для повторяющихся проверок лучше скомпилировать `Pattern` один раз и затем использовать `matcher.matches()` или `matcher.find()`.

```
// Если вам нужно часто проверять что-то простым regex:
boolean isValid = Pattern.matches("\\d+", inputString); // Для
одноразовой проверки нормально

// Если многократно:
private static final Pattern DIGITS_ONLY = Pattern.compile("\\d+");
// ... в цикле:
// boolean isValid = DIGITS_ONLY.matcher(inputString).matches(); //
Гораздо эффективнее
```

## 4.10. Создание шаблонов для сложных сценариев

Класс `Pattern` позволяет создавать шаблоны для очень сложных сценариев благодаря гибкости синтаксиса регулярных выражений.

- **Построение из нескольких частей**: Для очень длинных или трудночитаемых шаблонов можно объединять строковые константы, чтобы сделать их более модульными и понятными.

```
// Шаблон для IPv4-адреса
String octet = "([01]?\\d\\d?|2[0-4]\\d|25[0-5])"; // Одна октета (0-
255)
String ipAddressRegex = octet + "\\." + octet + "\\." + octet + "\\." +
octet;
Pattern ipPattern = Pattern.compile(ipAddressRegex);
```

- **Использование флага `Pattern.COMMENTS` (`(?x)`)**: Как обсуждалось ранее, этот флаг позволяет разбивать длинные Regex на несколько строк, добавлять пробелы и комментарии (`#`) для улучшения читаемости.

```
Pattern complexEmailPattern = Pattern.compile(
    "^[A-Z0-9._%+-]+" +  // Username (one or more A-Z0-9._%+- chars)
    "@" +                // At symbol
    "[A-Z0-9.-]+" +      // Domain name (one or more A-Z0-9.- chars)
    "\\." +              // Dot
    "[A-Z]{2,6}\$",       // Top-level domain (2 to 6 A-Z chars)
    Pattern.CASE_INSENSITIVE | Pattern.COMMENTS // Игнорировать
регистр, разрешить комментарии
);
```

```
// Пример с Pattern.COMMENTS, используя многострочный строковый литерал
(Java 15+)
// или конкатенацию строк (как показано выше)
Pattern formattedRegex = Pattern.compile("""
    ^                   # Start of the line
    (?:[a-z0-9!#\$%&'*+/=?^_`{|}~-]+(?:\\.[a-z0-9!#\$%&'*+/=?
^_`{|}~-]+)* # Username part
    |"(?:[\\x01-\\x08\\x0b\\x0c\\x0e-\\x1f\\x21\\x23-\\x5b\\x5d-
\\x7f]|\\\LEFTBRACKET\\x01-\\x09\\x0b\\x0c\\x0e-\\x7f])*" # Quoted
username part
    )@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.)+[a-z0-9](?:[a-z0-9-]*
[a-z0-9])? # Domain part
    |\LEFTBRACKET(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.){3}
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:(?:[\\x01-
\\x08\\x0b\\x0c\\x0e-\\x1f\\x21-\\x5a\\x5f-\\xff]|\\\LEFTBRACKET\\x01-
\\x09\\x0b\\x0c\\x0e-\\x7f])+)\RIGHTBRACKET # IP address literal
    )\$ # End of the line
    """, Pattern.CASE_INSENSITIVE | Pattern.COMMENTS);
```

Используя эти возможности класса `Pattern`, разработчики могут создавать надежные и эффективные решения для самых разнообразных задач обработки текста.

---

🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥

Отлично! Вот пятая глава, посвященная классу `Matcher` в Java:

---

# Глава 5: Класс `Matcher` в Java

## 5.1. Обзор класса `java.util.regex.Matcher`

Класс `java.util.regex.Matcher` является “движком” регулярных выражений в Java. В отличие от класса `Pattern`, который представляет скомпилированный шаблон, `Matcher` используется для выполнения конкретных операций сопоставления этого шаблона с входной строкой.

Объект `Matcher` сохраняет состояние процесса сопоставления, включая:

- Входную строку, с которой происходит сопоставление.
- Регулярное выражение (`Pattern`), которое используется.
- Текущую позицию, с которой будет начат следующий поиск.
- Информацию о последнем найденном совпадении (например, начало, конец, захваченные группы).

Поскольку `Matcher` хранит состояние, он **НЕ является потокобезопасным**. Каждый поток, которому нужно выполнить операцию сопоставления, должен использовать свой собственный экземпляр `Matcher`.

## 5.2. Создание объекта `Matcher` из `Pattern`

Объект `Matcher` всегда создается из объекта `Pattern` с помощью его метода `matcher()`.

**Синтаксис**:

```
public Matcher matcher(CharSequence input)
```

- `input`: Входная строка (`CharSequence` – это интерфейс, который `String` реализует), с которой будет сопоставляться шаблон.

**Пример (Java)**:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class MatcherCreationExample {
    public static void main(String[] args) {
        String text = "The year is 2024.";
        String regex = "\\d{4}"; // Шаблон для 4 цифр

        // 1. Компилируем регулярное выражение в объект Pattern
        Pattern pattern = Pattern.compile(regex);

        // 2. Создаем объект Matcher, передавая ему входную строку
        Matcher matcher = pattern.matcher(text);

        // Теперь можно использовать методы Matcher для поиска совпадений
        if (matcher.find()) {
            System.out.println("Found a 4-digit number: " +
matcher.group()); // 2024
        } else {
            System.out.println("No 4-digit number found.");
        }
    }
}
```

## 5.3. Метод `find()`: поиск следующего совпадения

Метод `find()` используется для поиска следующей подпоследовательности входной строки, которая соответствует шаблону.

**Синтаксис**:

```
public boolean find()
public boolean find(int start)
```

- `find()`: Пытается найти следующее совпадение, начиная с конца предыдущего совпадения (или с начала строки, если это первый вызов). Возвращает `true`, если найдено совпадение, `false` в противном случае.
- `find(int start)`: Пытается найти следующее совпадение, начиная с указанного индекса `start`.

Часто `find()` используется в цикле `while` для итерации по всем совпадениям во входной строке.

**Пример (Java)**:

```
public class MatcherFindExample {
    public static void main(String[] args) {
        String text = "Phone numbers: 123-456-7890, 987-654-3210. Email:
test@example.com.";
        Pattern pattern = Pattern.compile("\\d{3}-\\d{3}-\\d{4}");

        Matcher matcher = pattern.matcher(text);

        System.out.println("Finding all phone numbers:");
        while (matcher.find()) {
            System.out.println("  Found: " + matcher.group());
            // Output:
            //   Found: 123-456-7890
            //   Found: 987-654-3210
        }

        System.out.println("\nSearching from a specific index (starting
after 'Email:'):");
        // Предположим, мы знаем, что "Email: " начинается с индекса 35
        matcher.find(35); // Перемещаем указатель на позицию 35
        // Теперь ищем email
        Pattern emailPattern = Pattern.compile("\\b[A-Za-z0-9._%+-]+@[A-Za-
z0-9.-]+\\.[A-Z]{2,6}\\b", Pattern.CASE_INSENSITIVE);
        Matcher emailMatcher = emailPattern.matcher(text);

        // Метод find(int start) начинает поиск с указанной позиции.
        // Если предыдущий find() был успешен, следующий find() продолжит с
позиции после последнего совпадения.
        // Чтобы начать с конкретной позиции, можно использовать find(int
start) или reset(CharSequence input)
        // Для этого примера, давайте создадим новый Matcher или reset()
        emailMatcher.reset(text); // Сбросить matcher к началу строки
        if (emailMatcher.find(35)) { // Ищем email, начиная с индекса 35
            System.out.println("  Found email: " + emailMatcher.group());
// test@example.com
        } else {
            System.out.println("  No email found from index 35.");
        }
    }
}
```

## 5.4. Методы `matches()` и `lookingAt()`: различия и применение

Эти два метода также используются для проверки совпадений, но с разными условиями:

- `public boolean matches()`: Пытается сопоставить **всю** входную последовательность с шаблоном. Возвращает `true` только в том случае, если вся строка полностью соответствует регулярному выражению. Эквивалентно `^regex\$`.

- `public boolean lookingAt()`: Пытается сопоставить входную последовательность с шаблоном, **начиная с текущей позиции** matcher. Возвращает `true`, если начальная часть входной строки, начиная с текущей позиции, соответствует шаблону. Совпадение не обязательно должно охватывать всю входную строку.

**Пример (Java)**:

```
public class MatcherMatchesLookingAtExample {
    public static void main(String[] args) {
        String text1 = "abcdef";
        String text2 = "abcxyz";
        Pattern pattern = Pattern.compile("abc");

        Matcher matcher1 = pattern.matcher(text1);
        Matcher matcher2 = pattern.matcher(text2);

        // matches()
        System.out.println("Text '" + text1 + "' fully matches 'abc': " +
matcher1.matches()); // false (only 'abc' is matched, not 'abcdef')
        // Чтобы Pattern.matches() был true, regex должен быть
Pattern.compile("abcdef") или "abc.*"

        // lookingAt()
        // Reset matcher1 for lookingAt()
        matcher1.reset(text1);
        System.out.println("Text '" + text1 + "' starts with 'abc': " +
matcher1.lookingAt()); // true

        // Reset matcher2 for lookingAt()
        matcher2.reset(text2);
        System.out.println("Text '" + text2 + "' starts with 'abc': " +
matcher2.lookingAt()); // true

        // Сравнение matches() и lookingAt() с более полным примером
        String longText = "Start of text. The quick brown fox. End of
text.";
        Pattern wordPattern = Pattern.compile("\\bquick\\b");
        Matcher wordMatcher = wordPattern.matcher(longText);

        System.out.println("\nUsing 'quick' pattern on long text:");
        System.out.println("matches(): " + wordMatcher.matches()); // false
(because "quick" is not the entire string)
        wordMatcher.reset(longText); // Reset after previous operation
        System.out.println("lookingAt(): " + wordMatcher.lookingAt()); //
false (because "quick" is not at the beginning)
        wordMatcher.reset(longText); // Reset again
        System.out.println("find(): " + wordMatcher.find()); // true (it
will find "quick" somewhere in the middle)
        System.out.println("  Found: " + wordMatcher.group());
```

```
    }
}
```

## 5.5. Получение найденных групп: `group()`, `group(int)`, `groupCount()`

После успешного выполнения методов `find()`, `matches()` или `lookingAt()`, вы можете получить доступ к совпавшим подстрокам с помощью методов `group()`.

- `public String group()` или `public String group(0)`: Возвращает подстроку, которая совпала со **всем** регулярным выражением (группа 0).
- `public String group(int group)`: Возвращает подстроку, которая совпала с указанной захватывающей группой. Группы нумеруются слева направо, начиная с 1 (по порядку открывающих скобок).
- `public int groupCount()`: Возвращает количество захватывающих групп в шаблоне. (Это не включает группу 0).

**Пример (Java)**:

```
public class MatcherGroupExample {
    public static void main(String[] args) {
        String text = "Name: John Doe, Age: 30, City: New York";
        // Шаблон для извлечения имени, возраста и города
        Pattern pattern = Pattern.compile("Name: (\\w+\\s\\w+), Age: (\\d+),
City: (.*)");
        Matcher matcher = pattern.matcher(text);

        if (matcher.matches()) { // Используем matches(), чтобы убедиться,
что вся строка совпадает
            System.out.println("Full match (Group 0): " + matcher.group(0));
// Name: John Doe, Age: 30, City: New York
            System.out.println("Group count: " + matcher.groupCount());
// 3

            System.out.println("Name (Group 1): " + matcher.group(1));
// John Doe
            System.out.println("Age (Group 2): " + matcher.group(2));
// 30
            System.out.println("City (Group 3): " + matcher.group(3));
// New York
        } else {
            System.out.println("No full match found.");
        }

        // Пример с неполным совпадением и find()
        String text2 = "First part: A1B2C3, Second part: X9Y8Z7.";
        Pattern p2 = Pattern.compile("(\\w+): (\\w+)");
        Matcher m2 = p2.matcher(text2);

        System.out.println("\nFinding multiple parts:");
        while (m2.find()) {
```

```
            System.out.println("  Found section: " + m2.group(1) + " -> " +
m2.group(2));
            // Output:
            //   Found section: First -> A1B2C3
            //   Found section: Second -> X9Y8Z7
        }
    }
}
```

## 5.6. Определение позиций совпадений: `start()`, `end()`

Методы `start()` и `end()` возвращают индексы начала и конца (исключительно) найденного совпадения или захваченной группы.

- `public int start()` или `public int start(0)`: Возвращает начальный индекс совпавшего участка (для группы 0).
- `public int end()` или `public int end(0)`: Возвращает конечный индекс совпавшего участка (исключительно, для группы 0).
- `public int start(int group)`: Возвращает начальный индекс для указанной захваченной группы.
- `public int end(int group)`: Возвращает конечный индекс (исключительно) для указанной захваченной группы.

Если группа не участвовала в совпадении (например, если она была частью альтернативы, которая не совпала), `start()` и `end()` для этой группы вернут `-1`.

**Пример (Java)**:

```
public class MatcherPositionsExample {
    public static void main(String[] args) {
        String text = "The price is \$100.50, and quantity is 20.";
        Pattern pattern = Pattern.compile("\\$(\\d+\\.\\d+)"); // Извлекаем
число после \$

        Matcher matcher = pattern.matcher(text);

        if (matcher.find()) {
            System.out.println("Full match: " + matcher.group(0)); //
\$100.50
            System.out.println("  Start index of full match: " +
matcher.start(0)); // 13 (index of \$)
            System.out.println("  End index of full match: " +
matcher.end(0));     // 20 (index after 0)

            System.out.println("Captured group (Group 1): " +
matcher.group(1)); // 100.50
            System.out.println("  Start index of group 1: " +
matcher.start(1)); // 14 (index of 1)
            System.out.println("  End index of group 1: " + matcher.end(1));
// 20 (index after 0)
        }
```

```
    }
}
```

## 5.7. Методы замены: `replaceAll()` и `replaceFirst()`

Класс `Matcher` предоставляет удобные методы для замены текста, найденного по шаблону.

- `public String replaceAll(String replacement)`: Заменяет **каждое** совпадение шаблона в входной строке на указанную строку `replacement`.
- `public String replaceFirst(String replacement)`: Заменяет **первое** найденное совпадение шаблона на строку `replacement`.

В строке `replacement` можно использовать обратные ссылки на захваченные группы, используя синтаксис `\$n` (например, `\$1` для первой группы, `\$2` для второй).

**Пример (Java)**:

```
public class MatcherReplaceExample {
    public static void main(String[] args) {
        String text = "The date is 2024-08-17. Another date: 2025-01-01.";
        Pattern pattern = Pattern.compile("(\\d{4})-(\\d{2})-(\\d{2})"); //
yyyy-MM-dd

        // Замена первого совпадения
        Matcher matcher1 = pattern.matcher(text);
        String replacedFirst = matcher1.replaceFirst("DATE_FORMATTED");
        System.out.println("After replaceFirst: " + replacedFirst);
        // Output: The date is DATE_FORMATTED. Another date: 2025-01-01.

        // Замена всех совпадений с использованием обратных ссылок
(dd/MM/yyyy)
        Matcher matcher2 = pattern.matcher(text);
        String replacedAll = matcher2.replaceAll("\$3/\$2/\$1");
        System.out.println("After replaceAll (dd/MM/yyyy): " + replacedAll);
        // Output: The date is 17/08/2024. Another date: 01/01/2025.

        // Замена на текст, где используется literal "\$":
        String text3 = "Price: \$100.00";
        Pattern p3 = Pattern.compile("\\$(\\d+\\.\\d+)");
        Matcher m3 = p3.matcher(text3);
        // Чтобы вставить literal "\$" в строку замены, его нужно
экранировать: "\\$"
        String replacedWithLiteralDollar = m3.replaceAll("Cost: \\$\$1");
        System.out.println("Replaced with literal dollar: " +
replacedWithLiteralDollar); // Cost: \$100.00
    }
}
```

## 5.8. Построение замены с помощью `appendReplacement()` и `appendTail()`

Для более сложной логики замены, когда простая `replaceAll()` или `replaceFirst()` недостаточна, вы можете использовать `appendReplacement()` и `appendTail()`. Эти методы позволяют вам обрабатывать каждое совпадение по отдельности и строить результирующую строку шаг за шагом.

- `public Matcher appendReplacement(StringBuffer sb, String replacement)`: Добавляет в `StringBuffer` (или `StringBuilder`) часть входной строки до текущего совпадения, затем добавляет замену для текущего совпадения. Затем внутренний указатель `Matcher` перемещается за текущее совпадение.
- `public StringBuffer appendTail(StringBuffer sb)`: Добавляет в `StringBuffer` остаток входной строки после последнего совпадения. Этот метод вызывается один раз после того, как все совпадения обработаны с помощью `appendReplacement()`.

**Пример (Java)**:

```
import java.lang.StringBuffer; // Или StringBuilder для Java 5+

public class MatcherAppendReplacementExample {
    public static void main(String[] args) {
        String text = "Items: item1 (code A), item2 (code B), item3 (code
C).";
        Pattern pattern = Pattern.compile("item(\\d+) \LEFTPARENcode
(\\w)\RIGHTPAREN"); // itemN (code X)

        Matcher matcher = pattern.matcher(text);
        StringBuffer result = new StringBuffer(); // Используем StringBuffer
для потокобезопасности, хотя StringBuilder чаще для одиночных потоков

        while (matcher.find()) {
            String itemNumber = matcher.group(1);
            String itemCode = matcher.group(2);

            // Произвольная логика замены: меняем местами код и номер,
добавляем префикс
            String replacement = "CODE_" + itemCode + "_ITEM_" + itemNumber;

            // Добавляем текст до совпадения и замещенную часть
            matcher.appendReplacement(result, replacement);
        }
        // Добавляем остаток строки после последнего совпадения
        matcher.appendTail(result);

        System.out.println("Transformed text: " + result.toString());
        // Output: Items: CODE_A_ITEM_1, CODE_B_ITEM_2, CODE_C_ITEM_3.
    }
}
```

## 5.9. Метод `reset()` для повторного использования `Matcher`

Объект `Matcher` можно использовать повторно для новой входной строки или для повторного поиска в той же строке с самого начала.

- `public Matcher reset()`: Сбрасывает `Matcher` к его начальному состоянию. Указатель текущей позиции сбрасывается к 0, и информация о предыдущих совпадениях очищается. `Matcher` будет работать с той же входной строкой и шаблоном.
- `public Matcher reset(CharSequence input)`: Сбрасывает `Matcher` и привязывает его к новой входной строке `input`.

**Пример (Java)**:

```
public class MatcherResetExample {
    public static void main(String[] args) {
        String text1 = "The first number is 123.";
        String text2 = "The second number is 456.";
        Pattern pattern = Pattern.compile("\\d+");

        Matcher matcher = pattern.matcher(text1);

        // Первый поиск в text1
        if (matcher.find()) {
            System.out.println("First match in text1: " + matcher.group());
// 123
        }

        // Переиспользование matcher для новой строки
        matcher.reset(text2);
        if (matcher.find()) {
            System.out.println("First match in text2: " + matcher.group());
// 456
        }

        // Переиспользование matcher для той же строки, но сброс позиции для
повторного поиска
        matcher.reset(text1); // Снова привязываем к text1 и сбрасываем
состояние
        if (matcher.find()) {
            System.out.println("Re-found in text1 (after reset): " +
matcher.group()); // 123
        }
    }
}
```

## 5.10. Итерация по всем совпадениям в строке

Наиболее распространенный сценарий использования `Matcher` — это поиск всех вхождений шаблона в строке. Это обычно достигается с помощью цикла `while` в сочетании с методом `find()`.

**Пример (Java)**:

```
public class MatcherIterationExample {
    public static void main(String[] args) {
        String logEntry = "ERROR: Failed to connect (code 500). WARN: Low
```

```
disk space. INFO: User logged in. ERROR: Invalid input (code 400).";

        // Шаблон для поиска сообщений об ошибках с кодом
        Pattern errorPattern = Pattern.compile("ERROR: .*?\LEFTPARENcode
(\\d+)\RIGHTPAREN"); // Неленивый .*? для захвата всего до следующей скобки

        Matcher matcher = errorPattern.matcher(logEntry);

        System.out.println("Error codes found in log:");
        while (matcher.find()) {
            String fullError = matcher.group(0); // Все совпадение
            String errorCode = matcher.group(1); // Захваченная группа (код)

            System.out.println("  Full error message: \"" + fullError +
"\"");
            System.out.println("  Extracted error code: " + errorCode);
            System.out.println("  Start Index: " + matcher.start() + ", End
Index: " + matcher.end());
            System.out.println("---");
        }
    }
}
```

Эти методы класса `Matcher` составляют основу мощных возможностей Java по обработке и манипулированию текстом с использованием регулярных выражений.

---

🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥

Отлично! Вот шестая глава, посвященная практическим применениям регулярных выражений в Java:

---

# Глава 6: Практические Применения Регулярных Выражений в Java

## 6.1. Валидация форматов данных: общие подходы

Валидация данных — это процесс проверки того, соответствуют ли входные данные определенным правилам или форматам. Регулярные выражения являются одним из самых мощных и гибких инструментов для выполнения такой валидации.

**Общие подходы**:

1. **Полное совпадение**: Для большинства задач валидации требуется, чтобы **вся** строка соответствовала шаблону. Для этого обычно используют `Pattern.matches(regex, input)` или `pattern.matcher(input).matches()`. Это гарантирует, что нет лишних символов до или после ожидаемого формата.
2. **Точность шаблона**: Шаблон должен быть достаточно точным, чтобы отсеять некорректные данные, но при этом достаточно гибким, чтобы принимать все действительные варианты. Это часто компромисс между строгостью и удобством использования.
3. **Использование границ**: Использование якорей `^` (начало строки) и `\$` (конец строки) критически важно для валидации, чтобы шаблон соответствовал всей строке, а не только

её части.
4. **Обработка ошибок**: Важно не только знать, что данные невалидны, но и, возможно, предоставить обратную связь пользователю о том, какой формат ожидается.

**Пример (Java)**:

```
import java.util.regex.Pattern;

public class DataValidationGeneral {
    public static void main(String[] args) {
        String input1 = "12345";
        String input2 = "1234";
        String input3 = "12345abc";

        // Проверяем, состоит ли строка ровно из 5 цифр
        String regex = "^\\d{5}\$";

        System.out.println("Input '" + input1 + "' is valid: " +
Pattern.matches(regex, input1)); // true
        System.out.println("Input '" + input2 + "' is valid: " +
Pattern.matches(regex, input2)); // false
        System.out.println("Input '" + input3 + "' is valid: " +
Pattern.matches(regex, input3)); // false
    }
}
```

## 6.2. Пример: Валидация email-адресов

Валидация email-адресов — классический пример использования регулярных выражений. Однако стоит отметить, что создание *идеального* регулярного выражения для email, соответствующего всем стандартам RFC, чрезвычайно сложно и приводит к очень длинным и нечитаемым шаблонам. На практике обычно используются более простые, но достаточно надежные Regex, которые охватывают большинство распространенных форматов.

**Распространенный (но не исчерпывающий) шаблон для email**: `^[\\w-\\.]+@([\\w-]+\\.)+[\\w-]{2,4}\$`

- `^`: Начало строки.
- `[\\w-\\.]+`: Имя пользователя: одна или более букв, цифр, _, -, . (но не в начале/конце и не подряд).
- `@`: Символ “собачки”.
- `([\\w-]+\\.)+`: Доменное имя: одна или более подчастей домена (например, `example.`), где каждая подчасть состоит из букв, цифр, -, заканчивается на `.`. `+` указывает на одну или более таких подчастей (например, `sub.domain.`).
- `[\\w-]{2,4}`: Домен верхнего уровня (TLD): 2-4 символа (буквы, цифры, -).
- `\$`: Конец строки.

**Пример (Java)**:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class EmailValidator {
```

```
    // Шаблон достаточно общий для большинства случаев
    private static final Pattern EMAIL_PATTERN =
        Pattern.compile("^[\\w!#\$%&'*+/=?`{|}~^-]+(?:\\.[\\w!#\$%&'*+/=?
`{|}~^-]+)*@(?:[a-zA-Z0-9-]+\\.)+[a-zA-Z]{2,6}\$",
                        Pattern.CASE_INSENSITIVE); // Регистронезависимый
для домена

    public static boolean isValidEmail(String email) {
        return EMAIL_PATTERN.matcher(email).matches();
    }

    public static void main(String[] args) {
        String[] emails = {
            "test@example.com",
            "john.doe123@sub.domain.co.uk",
            "invalid-email",
            "user@domain",
            "user@.com",
            "name@company.museum", // >4 chars TLD
            "another_user@mail.info" // >4 chars TLD
        };

        for (String email : emails) {
            System.out.println("'" + email + "' is valid: " +
isValidEmail(email));
        }
        // Уточнение: для TLD можно использовать {2,6} или {2,} для большей
гибкости
        // Для TLD [a-zA-Z]{2,6} может быть недостаточным, так как есть TLD
длиннее 6 символов.
        // Более точный TLD: [a-zA-Z]{2,}
    }
}
```

## 6.3. Пример: Валидация телефонных номеров

Форматы телефонных номеров сильно различаются в разных странах. Для примера рассмотрим простой американский формат: `(XXX) XXX-XXXX` или `XXX-XXX-XXXX`.

**Шаблон**: `^\LEFTPAREN?\\d{3}\RIGHTPAREN?[- ]?\\d{3}[- ]?\\d{4}\$`

- `^`: Начало строки.
- `\LEFTPAREN?`: Необязательная открывающая скобка.
- `\\d{3}`: Три цифры.
- `\RIGHTPAREN?`: Необязательная закрывающая скобка.
- `[- ]?`: Необязательный дефис или пробел.
- `\\d{3}`: Еще три цифры.
- `[- ]?`: Необязательный дефис или пробел.
- `\\d{4}`: Четыре цифры.
- `\$`: Конец строки.

**Пример (Java)**:

```
public class PhoneNumberValidator {
    private static final Pattern PHONE_PATTERN =
        Pattern.compile("^\LEFTPAREN?\\d{3}\RIGHTPAREN?[- ]?\\d{3}[- ]?
\\d{4}\$");

    public static boolean isValidPhoneNumber(String phoneNumber) {
        return PHONE_PATTERN.matcher(phoneNumber).matches();
    }

    public static void main(String[] args) {
        String[] phones = {
            "123-456-7890",
            "(123) 456-7890",
            "123 456 7890",
            "(123)456-7890",
            "1234567890", // Совпадет, т.к. разделители необязательны
            "123-456-789", // Слишком короткий
            "123-456-78901", // Слишком длинный
            "abc-def-ghij"
        };

        for (String phone : phones) {
            System.out.println("'" + phone + "' is valid: " +
isValidPhoneNumber(phone));
        }
    }
}
```

## 6.4. Пример: Валидация URL-адресов и IP-адресов

**Валидация URL-адресов**: Также сложная задача для идеального охвата всех стандартов. Приведем упрощенный шаблон.

**Шаблон URL**: `^(https?|ftp|file)://[-a-zA-Z0-9+&@#/%?=~_|!:,.;]*[-a-zA-Z0-9+&@#/%=~_|]\$` (Это базовый шаблон, который не охватывает все возможные символы и синтаксис URL, но подойдет для большинства случаев.)

**Пример URL (Java)**:

```
public class UrlValidator {
    // Упрощенный, но рабочий шаблон URL
    private static final Pattern URL_PATTERN =
        Pattern.compile("^(https?|ftp|file)://[-a-zA-Z0-9+&@#/%?=~_|!:,.;]*
[-a-zA-Z0-9+&@#/%=~_|]\$");

    public static boolean isValidUrl(String url) {
        return URL_PATTERN.matcher(url).matches();
    }
```

```
    public static void main(String[] args) {
        String[] urls = {
            "http://www.example.com",
            "https://sub.domain.com/path/to/page.html?param=value#anchor",
            "ftp://files.server.org/data.zip",
            "file:///C:/Users/User/document.pdf",
            "invalid-url",
            "http://example.com/a b c" // Пробелы не допускаются без
кодирования
        };

        for (String url : urls) {
            System.out.println("'" + url + "' is valid: " +
isValidUrl(url));
        }
    }
}
```

**Валидация IP-адресов (IPv4)**: IP-адрес состоит из четырех октетов, разделенных точками, каждый октет — число от 0 до 255.

**Шаблон IPv4**: `^([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\." +([01]?\d\d?|2[0-4]\d|25[0-5])\." + ([01]?\\d\\d?|2[0-4]\\d|25[0-5])\\." +([01]?\d\d?|2[0-4]\d|25[0-5])$`

- `([01]?\\d\\d?|2[0-4]\\d|25[0-5])`: Это группа, которая сопоставляет число от 0 до 255.
  - `[01]?\\d\\d?`: 0-9, 00-99, 100-199 (т.е. 0-199).
  - `2[0-4]\\d`: 200-249.
  - `25[0-5]`: 250-255.
- `\\.`: Буквальная точка.

**Пример IPv4 (Java)**:

```
public class IpAddressValidator {
    // Шаблон для одного октета (0-255)
    private static final String OCTET_REGEX = "([01]?\\d\\d?|2[0-4]\\d|25[0-
5])";
    private static final Pattern IPV4_PATTERN =
        Pattern.compile("^" + OCTET_REGEX + "\\." + OCTET_REGEX + "\\." +
OCTET_REGEX + "\\." + OCTET_REGEX + "\$");

    public static boolean isValidIPv4(String ip) {
        return IPV4_PATTERN.matcher(ip).matches();
    }

    public static void main(String[] args) {
        String[] ips = {
            "192.168.1.1",
            "0.0.0.0",
            "255.255.255.255",
            "10.0.0.1",
            "192.168.1.256", // Некорректно
```

```
            "1.2.3", // Некорректно
            "255.255.255.255.", // Некорректно
            "192.168.1.1.1" // Некорректно
        };

        for (String ip : ips) {
            System.out.println("'" + ip + "' is valid IPv4: " +
isValidIPv4(ip));
        }
    }
}
```

## 6.5. Парсинг и извлечение данных из текстовых файлов (логи, конфиги)

Регулярные выражения отлично подходят для извлечения структурированных данных из неструктурированного текста, такого как логи или конфигурационные файлы.

**Пример: Извлечение информации из лог-файла** Предположим, у нас есть лог-файл со строками в формате: `[YYYY-MM-DD HH:MM:SS] [LEVEL] Message text. (User: <username>, ID: <id>)`

**Шаблон**: `^\LEFTBRACKET(\\d{4}-\\d{2}-\\d{2} \\d{2}:\\d{2}:\\d{2})\RIGHTBRACKET \LEFTBRACKET([A-Z]+)\RIGHTBRACKET (.*?)(?: \LEFTPARENUser: (\\w+), ID: (\\d+)\RIGHTPAREN)?\$`

- `^`: Начало строки.
- `\LEFTBRACKET(.*?)\RIGHTBRACKET`: Захватывает содержимое внутри квадратных скобок.
- `(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2})`: Группа 1 (Дата и время).
- `([A-Z]+)`: Группа 2 (Уровень лога: INFO, WARN, ERROR).
- `(.*?)`: Группа 3 (Текст сообщения, ленивый).
- `(?: \LEFTPARENUser: (\\w+), ID: (\\d+)\RIGHTPAREN)?`: Необязательная незахватывающая группа для информации о пользователе.
  - `(\\w+)`: Группа 4 (Имя пользователя).
  - `(\\d+)`: Группа 5 (ID пользователя).

**Пример (Java)**:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class LogParser {
    private static final Pattern LOG_PATTERN =
        Pattern.compile("^\LEFTBRACKET(\\d{4}-\\d{2}-\\d{2}
\\d{2}:\\d{2}:\\d{2})\RIGHTBRACKET \LEFTBRACKET([A-Z]+)\RIGHTBRACKET (.*?)
(?: \LEFTPARENUser: (\\w+), ID: (\\d+)\RIGHTPAREN)?\$");

    public static void main(String[] args) {
        String logEntry1 = "[2024-08-17 10:30:00] [INFO] Application started
successfully.";
        String logEntry2 = "[2024-08-17 10:35:15] [ERROR] Invalid login
attempt. (User: admin, ID: 123)";
        String logEntry3 = "[2024-08-17 10:40:05] [WARN] Disk space low on
drive C:";
```

```
        String[] logEntries = {logEntry1, logEntry2, logEntry3};

        for (String entry : logEntries) {
            Matcher matcher = LOG_PATTERN.matcher(entry);
            if (matcher.matches()) {
                System.out.println("--- Log Entry ---");
                System.out.println("Timestamp: " + matcher.group(1));
                System.out.println("Level:     " + matcher.group(2));
                System.out.println("Message:   " + matcher.group(3));
                if (matcher.group(4) != null) { // Проверяем, существует ли
группа пользователя
                    System.out.println("User:      " + matcher.group(4));
                    System.out.println("User ID:   " + matcher.group(5));
                }
            } else {
                System.out.println("Could not parse log entry: " + entry);
            }
        }
    }
}
```

## 6.6. Поиск и замена текста в строках

Методы `replaceAll()` и `replaceFirst()` класса `Matcher` невероятно полезны для массового изменения текста.

**Пример: Нормализация дат из `YYYY-MM-DD` в `DD.MM.YYYY`**

```
public class DateFormatter {
    private static final Pattern DATE_PATTERN =
        Pattern.compile("(\\d{4})-(\\d{2})-(\\d{2})"); // Группа 1: год, 2:
месяц, 3: день

    public static String normalizeDate(String text) {
        Matcher matcher = DATE_PATTERN.matcher(text);
        // Используем обратные ссылки \$3 (день), \$2 (месяц), \$1 (год)
        return matcher.replaceAll("\$3.\$2.\$1");
    }

    public static void main(String[] args) {
        String article = "Today's date is 2024-08-17. The project deadline
was 2023-12-31.";
        String formattedArticle = normalizeDate(article);
        System.out.println("Original:  " + article);
        System.out.println("Formatted: " + formattedArticle);
        // Output: Formatted: Today's date is 17.08.2024. The project
deadline was 31.12.2023.
```

```
    }
}
```

## 6.7. Разделение строк на токены или компоненты

Метод `Pattern.split()` позволяет легко разбивать строки на массив подстрок по заданному регулярному выражению-разделителю.

**Пример: Разделение строки CSV с различными разделителями и пробелами**

```
public class CSVSplitter {
    // Разделители: запятая, точка с запятой, табуляция или один/более
пробелов
    private static final Pattern DELIMITER_PATTERN = Pattern.compile("
[,;\\t\\s]+");

    public static String[] splitCSV(String line) {
        return DELIMITER_PATTERN.split(line);
    }

    public static void main(String[] args) {
        String dataLine1 = "apple,banana;orange   grape";
        String dataLine2 = "data1\tdata2,data3;data4  data5";

        System.out.println("Splitting line 1:");
        for (String item : splitCSV(dataLine1)) {
            System.out.println("  \"" + item + "\"");
        }

        System.out.println("\nSplitting line 2:");
        for (String item : splitCSV(dataLine2)) {
            System.out.println("  \"" + item + "\"");
        }
    }
}
```

## 6.8. Массовое переименование файлов

Хотя прямое переименование файлов в Java не является частью `java.util.regex`, регулярные выражения часто используются для формирования новых имен файлов на основе старых. Вы можете скомбинировать Regex с классами `java.io.File` (или `java.nio.file.Path` для современной работы с файлами).

**Концептуальный пример: Переименование файлов из `report_YYYYMMDD.txt` в `YYYY-MM-DD_report.txt`**

```
import java.io.File;
import java.nio.file.Files;
import java.nio.file.Path;
```

```
import java.nio.file.Paths;
import java.io.IOException;

public class FileRenamer {
    // Шаблон для имен файлов: report_YYYYMMDD.txt
    private static final Pattern FILENAME_PATTERN =
        Pattern.compile("report_(\\d{4})(\\d{2})(\\d{2})\\.txt");

    public static void renameFiles(String directoryPath) {
        File directory = new File(directoryPath);
        File[] files = directory.listFiles();

        if (files == null) {
            System.out.println("Directory not found or is empty: " +
directoryPath);
            return;
        }

        for (File file : files) {
            String oldName = file.getName();
            Matcher matcher = FILENAME_PATTERN.matcher(oldName);

            if (matcher.matches()) {
                String year = matcher.group(1);
                String month = matcher.group(2);
                String day = matcher.group(3);

                // Формируем новое имя: YYYY-MM-DD_report.txt
                String newName = year + "-" + month + "-" + day +
"_report.txt";

                Path oldPath = file.toPath();
                Path newPath = Paths.get(directoryPath, newName);

                try {
                    Files.move(oldPath, newPath);
                    System.out.println("Renamed '" + oldName + "' to '" +
newName + "'");
                } catch (IOException e) {
                    System.err.println("Failed to rename " + oldName + ": "
+ e.getMessage());
                }
            }
        }
    }

    public static void main(String[] args) {
        // Создайте тестовую папку и файлы для проверки:
        // C:/temp/my_reports/report_20240101.txt
```

```
        // C:/temp/my_reports/report_20240817.txt
        // C:/temp/my_reports/other_file.log

        //renameFiles("C:/temp/my_reports"); // Замените на реальный путь
        System.out.println("Please uncomment and set the correct directory
path to run file renaming example.");
    }
}
```

## 6.9. Обработка данных из HTML/XML (с оговорками)

**Важное предупреждение: Использование регулярных выражений для парсинга HTML или XML-документов является плохой практикой и часто приводит к неверным результатам и уязвимостям.** HTML/XML — это иерархические, нерегулярные (в смысле регулярных языков) структуры, которые не могут быть адекватно проанализированы регулярными выражениями. Для этой цели следует использовать специализированные парсеры (например, Jsoup для HTML, DOM/SAX парсеры для XML в Java).

Тем не менее, для **очень простых и специфичных** задач, где структура гарантированно проста и неизменна, Regex может быть временно использован.

**Пример (с оговорками): Извлечение всех `href` атрибутов из простых `<a>` тегов**

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class HtmlSimpleExtractor {
    // ОЧЕНЬ УПРОЩЕННЫЙ шаблон для href (НЕ ИСПОЛЬЗУЙТЕ В ПРОДАКШЕНЕ!)
    private static final Pattern HREF_PATTERN = Pattern.compile("
<a\\s+href=\"(.*?)\"");

    public static void extractHrefs(String html) {
        Matcher matcher = HREF_PATTERN.matcher(html);
        System.out.println("Extracting hrefs (DANGER: simplistic
approach!):");
        while (matcher.find()) {
            System.out.println("  Found href: " + matcher.group(1));
        }
    }

    public static void main(String[] args) {
        String htmlContent = "<p>Visit our <a
href=\"https://www.example.com\">website</a> or check <a
href=\"/products\">our products</a>.</p>";
        extractHrefs(htmlContent);

        // Пример, почему это плохо:
        String badHtml = "<a title=\"href=\'bad\'\"
href=\"good.html\">Link</a>";
        System.out.println("\n--- BAD EXAMPLE ---");
```

```
        extractHrefs(badHtml); // Выведет "good.html", но если бы title был
длиннее, могли бы быть проблемы.
                               // Игнорирует атрибуты до href, что может быть
неверным
                               // Не обрабатывает двойные кавычки внутри href
или другие сложные сценарии
    }
}
```

## 6.10. Задачи по обработке естественного языка (NLP): токенизация, выделение сущностей

В NLP регулярные выражения используются для базовых задач, таких как токенизация (разделение текста на слова, предложения) или извлечение простых именованных сущностей (например, числа, даты, email-адреса). Для более сложных NLP задач используются специализированные библиотеки (Apache OpenNLP, Stanford CoreNLP).

**Пример: Простая токенизация (разделение на слова)**

```
public class SimpleTokenizer {
    // Разделяем по всем непробельным символам (т.е. получаем слова)
    private static final Pattern WORD_DELIMITER =
Pattern.compile("\\P{L}+"); // \P{L} - любой символ, НЕ буква

    public static String[] tokenize(String text) {
        // Мы используем Pattern.split, но также можно найти слова:
\\b\\w+\\b
        return WORD_DELIMITER.split(text);
    }

    public static void main(String[] args) {
        String sentence = "Hello, world! This is a test sentence.";
        String[] words = tokenize(sentence);

        System.out.println("Tokens in the sentence:");
        for (String word : words) {
            if (!word.isEmpty()) { // split может оставлять пустые строки,
если разделители в начале/конце/подряд
                System.out.println("  \"" + word + "\"");
            }
        }
        // Output:
        //   "Hello"
        //   "world"
        //   "This"
        //   "is"
        //   "a"
        //   "test"
        //   "sentence"
```

```
        // Альтернативный подход: находить слова
        Pattern wordFinder = Pattern.compile("\\b\\w+\\b");
        Matcher matcher = wordFinder.matcher(sentence);
        System.out.println("\nWords found using find():");
        while(matcher.find()) {
            System.out.println("  \"" + matcher.group() + "\"");
        }
    }
}
```

Эти примеры демонстрируют широкие возможности регулярных выражений в Java для решения повседневных и более сложных задач по обработке текста.

---

🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥

Отлично! Вот седьмая глава, посвященная производительности и оптимизации регулярных выражений в Java:

---

# Глава 7: Производительность и Оптимизация Regex в Java

## 7.1. Влияние регулярных выражений на производительность

Регулярные выражения — мощный инструмент, но их неправильное использование может привести к значительным проблемам с производительностью, особенно при работе с большими объемами текста или в высоконагруженных системах. Производительность регулярных выражений зависит от нескольких факторов:

- **Сложность шаблона**: Чем сложнее и нечетче шаблон (например, слишком много альтернатив, вложенных групп, или использование `.*`), тем больше работы приходится выполнять движку Regex.
- **Длина входной строки**: Время выполнения Regex обычно возрастает с длиной входной строки. В некоторых случаях (при “катастрофическом откате”) это увеличение может быть экспоненциальным.
- **Характеристики движка Regex**: Разные движки (Java, Perl, Python) имеют свои особенности оптимизации. Java `java.util.regex` достаточно оптимизирован, но не все шаблоны обрабатываются одинаково эффективно.
- **Количество совпадений**: Если шаблон часто встречается в тексте, `find()` будет вызываться много раз, что может быть быстрее, чем поиск единственного, но очень сложного шаблона.

Особое внимание следует уделять случаям, когда Regex используется в циклах или обрабатывает данные от ненадежных источников, так как это может привести к атакам типа ReDoS (Regular Expression Denial of Service).

## 7.2. Проблема “катастрофического отката” (catastrophic backtracking) и как ее избежать

Катастрофический откат — это одна из самых серьезных проблем производительности в регулярных выражениях. Она возникает, когда шаблон содержит повторяющиеся группы, которые могут сопоставлять одни и те же подстроки несколькими способами, и при этом в конце шаблона находится часть, которая не совпадает. Движок Regex вынужден пробовать огромное

количество комбинаций, откатываясь (backtracking) и перебирая варианты, что может привести к экспоненциальному росту времени выполнения.

**Пример шаблона, подверженного катастрофическому откату**: `(a+)+` или `(ab|a)*c` Например, для `(a+)+` и строки `"aaaaaaaaaaaaaaaaaaaX"`:

- `a+` может сопоставить любую подпоследовательность `a`.
- Внешний `+` также может сопоставить одну или несколько последовательностей `a+`. Движок будет пытаться выделить `a+` самым длинным способом, затем откатываться, чтобы внешний `+` мог найти `a+` из оставшихся `a`, и так далее, пока не достигнет `X`. Тогда он начнет откатываться по всем комбинациям `a+` внутри `(a+)+`, что приводит к экспоненциальной сложности.

**Как избежать**:

1. **Используйте атомарные группы `(?>...)` или сверхагрессивные квантификаторы `*+`, `++`, `?+`**: Они отключают откат внутри группы, заставляя движок “фиксировать” совпадение и не возвращаться к нему.
2. **Избегайте избыточных повторений**: Например, `(a*)*` почти всегда можно заменить на `a*`.
3. **Используйте точные символьные классы**: Вместо `.` используйте `\d`, `\w`, если это возможно.
4. **Упрощайте шаблон**: Часто более простой, но чуть менее общий шаблон работает намного быстрее.
5. **Используйте негативные опережающие или ретроспективные проверки (Negative Lookahead/Lookbehind)**, чтобы исключить нежелательные пути сопоставления.

**Пример (Java)**:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class CatastrophicBacktrackingExample {
    public static void main(String[] args) {
        String data = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaX"; // Длинная строка
'a' и один 'X'

        // Плохой шаблон: (a+)+ - подвержен катастрофическому откату
        Pattern badPattern = Pattern.compile("(a+)+X");
        long start = System.nanoTime();
        Matcher badMatcher = badPattern.matcher(data);
        System.out.println("Bad pattern matches: " + badMatcher.matches());
        System.out.println("Time for bad pattern: " + (System.nanoTime() -
start) / 1_000_000.0 + " ms"); // Может быть очень долго

        // Хороший шаблон: a+X - эквивалентен, но без отката
        Pattern goodPattern1 = Pattern.compile("a+X");
        start = System.nanoTime();
        Matcher goodMatcher1 = goodPattern1.matcher(data);
        System.out.println("Good pattern 1 matches: " +
goodMatcher1.matches());
        System.out.println("Time for good pattern 1: " + (System.nanoTime()
- start) / 1_000_000.0 + " ms"); // Намного быстрее

        // Хороший шаблон: (?>a+)+X - использование атомарной группы
        Pattern goodPattern2 = Pattern.compile("(?>a+)+X");
```

```
        start = System.nanoTime();
        Matcher goodMatcher2 = goodPattern2.matcher(data);
        System.out.println("Good pattern 2 (atomic) matches: " +
goodMatcher2.matches());
        System.out.println("Time for good pattern 2: " + (System.nanoTime()
- start) / 1_000_000.0 + " ms"); // Также быстро
    }
}
```

## 7.3. Выбор между жадными, ленивыми и сверхагрессивными квантификаторами для оптимизации

Выбор правильного типа квантификатора может существенно повлиять на производительность и корректность сопоставления.

- **Жадные (Greedy)**: `*`, `+`, `?`, `{n,m}`. Сопоставляют как можно больше символов. Могут вызвать откат, если остальная часть шаблона не может быть сопоставлена.
    - **Применение**: Когда вам нужно захватить максимально длинное совпадение.
    - **Производительность**: Хорошо, если совпадение находится быстро. Плохо, если много отката.
- **Ленивые (Reluctant/Lazy)**: `*?`, `+?`, `??`, `{n,m}?`. Сопоставляют как можно меньше символов. Движок будет расширять совпадение только по мере необходимости.
    - **Применение**: Когда вам нужно захватить минимально длинное совпадение, или когда жадный квантификатор захватывает слишком много (например, `.*` внутри HTML-тегов).
    - **Производительность**: Могут быть медленнее жадных, если им приходится часто “расширяться” по одному символу, но часто предотвращают катастрофический откат в определенных сценариях (например, `.*?` для сопоставления между ограничителями).
- **Сверхагрессивные (Possessive)**: `*+`, `++`, `?+`, `{n,m}+`. Ведут себя как жадные, но **не допускают отката**. Если сверхагрессивный квантификатор захватывает текст, и это не позволяет всему шаблону совпасть, то совпадение не будет найдено, даже если откат мог бы привести к успеху.
    - **Применение**: Когда вы точно знаете, что откат для определенной части шаблона не нужен и только потратит время.
    - **Производительность**: Самые быстрые, когда они применимы, потому что они исключают откат. Используйте с осторожностью, так как могут пропустить действительные совпадения, если ваша логика отката критична.

**Пример (Java)**: См. пример в разделе 2.10 для сравнения жадных, ленивых и сверхагрессивных квантификаторов и их влияния на совпадение.

## 7.4. Использование атомарных групп для предотвращения отката

Атомарные группы `(?>pattern)` — это частный случай незахватывающих групп, которые также отключают откат для шаблона, который они содержат. Как только атомарная группа успешно сопоставляется с некоторой частью входной строки, эта часть “фиксируется”, и движок регулярных выражений не будет пытаться откатиться и переоценить ее, даже если это необходимо для успешного сопоставления оставшейся части шаблона.

**Преимущества**:

- **Улучшение производительности**: Предотвращает избыточный откат, который может быть катастрофическим.
- **Изменение поведения сопоставления**: Может привести к тому, что шаблон не найдет совпадение там, где он нашел бы его с жадными или ленивыми квантификаторами (как показано в примере `(a|ab)+a` vs `(?>a|ab)+a` в Главе 3).

**Когда использовать**:

- Когда вы уверены, что если внутренняя часть группы совпала, то она должна быть зафиксирована, и дальнейший откат бесполезен.
- Для борьбы с катастрофическим откатом.

**Пример (Java)**: См. пример в разделе 3.6 для демонстрации атомарных групп.

## 7.5. Компиляция шаблонов один раз: `Pattern.compile()`

Это, пожалуй, наиболее фундаментальная и важная оптимизация при работе с регулярными выражениями в Java. Как обсуждалось в Главе 4, процесс компиляции строкового регулярного выражения в объект `Pattern` является ресурсоемким. Если вы используете один и тот же шаблон несколько раз, его следует скомпилировать только один раз.

**Рекомендации**:

- Объявляйте объекты `Pattern` как `static final` поля класса, если они используются в нескольких местах или методах.
- Если шаблон нужен только в одном методе, но вызывается многократно, скомпилируйте его один раз вне цикла.

**Пример (Java)**:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class CompileOnceExample {

    // Правильный подход: Компиляция один раз
    private static final Pattern VALID_EMAIL_PATTERN =
        Pattern.compile("^[A-Z0-9._%+-]+@[A-Z0-9.-]+\\.[A-Z]{2,6}\$",
Pattern.CASE_INSENSITIVE);

    public boolean checkEmailEfficiently(String email) {
        return VALID_EMAIL_PATTERN.matcher(email).matches();
    }

    // Плохой подход: Компиляция при каждом вызове
    public boolean checkEmailInefficiently(String email) {
        // Каждый раз создается новый Pattern
        return Pattern.compile("^[A-Z0-9._%+-]+@[A-Z0-9.-]+\\.[A-Z]{2,6}\$",
Pattern.CASE_INSENSITIVE).matcher(email).matches();
    }

    public static void main(String[] args) {
        CompileOnceExample example = new CompileOnceExample();
        String testEmail = "test@example.com";
```

```
        int iterations = 1_000_000;

        long startTime = System.nanoTime();
        for (int i = 0; i < iterations; i++) {
            example.checkEmailEfficiently(testEmail);
        }
        long endTime = System.nanoTime();
        System.out.println("Efficient check (" + iterations + " iterations):
" + (endTime - startTime) / 1_000_000.0 + " ms");

        startTime = System.nanoTime();
        for (int i = 0; i < iterations; i++) {
            example.checkEmailInefficiently(testEmail);
        }
        endTime = System.nanoTime();
        System.out.println("Inefficient check (" + iterations + "
iterations): " + (endTime - startTime) / 1_000_000.0 + " ms");
        // Разница будет ОЧЕНЬ существенной (от десятков до сотен раз).
    }
}
```

## 7.6. Когда избегать Regex: альтернативы для простых задач

Хотя регулярные выражения мощны, они не всегда являются лучшим решением. Для простых задач стандартные строковые методы Java могут быть более читаемыми и производительными.

**Примеры, когда Regex избыточен**:

- **Проверка наличия подстроки**: Используйте `String.contains()` или `String.indexOf()`.

```
String text = "Hello World";
// Вместо Pattern.compile("World").matcher(text).find()
System.out.println(text.contains("World")); // true
```

- **Проверка, начинается ли строка с/заканчивается на**: Используйте `String.startsWith()` и `String.endsWith()`.

```
String filename = "document.txt";
// Вместо Pattern.matches(".*\\.txt\$", filename)
System.out.println(filename.endsWith(".txt")); // true
```

- **Разделение строк по фиксированному разделителю**: Используйте `String.split()`. (Хотя он принимает Regex, для простых разделителей нет необходимости в полной Regex-системе).

```
String csv = "apple,banana,orange";
// Вместо Pattern.compile(",").split(csv)
String[] items = csv.split(",");
```

- **Простая замена подстроки**: Используйте `String.replace()` или `String.replaceFirst()`/`replaceAll()` (они также принимают literal `CharSequence` или `String`).

```
String sentence = "Hello Java. Hello World.";
// Вместо Pattern.compile("Hello").matcher(sentence).replaceAll("Hi")
System.out.println(sentence.replace("Hello", "Hi")); // Hi Java. Hi
World.
```

Использование встроенных строковых методов часто приводит к более понятному коду и лучшей производительности, поскольку они оптимизированы для своих специфических задач.

## 7.7. Оптимизация сложных регулярных выражений: пошаговый подход

Для оптимизации сложных Regex, которые не страдают от катастрофического отката, но все равно медленные, можно использовать пошаговый подход:

1. **Профилирование**: Всегда начинайте с профилирования, чтобы убедиться, что именно Regex является узким местом, и определить, какие части шаблона потребляют больше всего времени (см. 7.8).
2. **Декомпозиция**: Разделите очень длинные и сложные шаблоны на несколько более простых, если это возможно. Возможно, часть валидации можно выполнить простыми строковыми методами, а затем применить Regex к оставшейся части.
3. **Специфичность вместо обобщенности**:
   - Используйте `\d`, `\w`, `\s` вместо `.` там, где это уместно.
   - Используйте конкретные диапазоны в символьных классах (например, `[0-9]` вместо `\d`, если вам нужны только ASCII цифры, или `[a-zA-Z]` вместо `\w` если нужны только буквы).
4. **Уточняйте квантификаторы**:
   - Если знаете точное количество повторений, используйте `{n}`.
   - Если есть максимум, используйте `{n,m}`.
   - Переключайтесь на ленивые или сверхагрессивные квантификаторы, если жадные приводят к ненужному откату (и профилирование это подтверждает).
5. **Избегайте ненужных групп**: Используйте незахватывающие группы `(?:...)`, если вам не нужны обратные ссылки.
6. **Якоря**: Используйте `^` и `\$` (или `\A`, `\Z`, `\z` для абсолютного начала/конца входных данных), чтобы помочь движку быстрее отбросить несоответствующие строки или избежать ненужных поисков в середине строки.
7. **Оптимизация альтернатив**: В `(A|B|C)`, поместите наиболее часто встречающиеся альтернативы первыми.
8. **Используйте флаги**: `Pattern.CASE_INSENSITIVE` может быть медленнее, чем ручная обработка регистров, но часто удобнее. `Pattern.COMMENTS` (`(?x)`) не влияет на производительность выполнения, только на читаемость.

## 7.8. Профилирование производительности Regex-кода

Профилирование позволяет измерить, сколько времени и ресурсов потребляют различные части вашей программы, включая Regex.

**Инструменты профилирования в Java**:

- **Встроенные JDK инструменты**: `jvisualvm` (Java VisualVM) может подключаться к запущенным JVM и предоставлять информацию о производительности CPU, памяти, потоках и т.д. Он может показать, какие методы потребляют больше всего CPU времени.
- **Коммерческие/профессиональные профилировщики**: YourKit, JProfiler, IntelliJ IDEA Ultimate's Profiler. Они предлагают более детальный анализ, включая время выполнения Regex-операций.
- **Ручной бенчмаркинг**: Для быстрого замера можно использовать `System.nanoTime()` до и после выполнения Regex-операции.

**Пример ручного бенчмаркинга (Java)**:

```java
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegexBenchmarking {
    public static void main(String[] args) {
        String longText = "This is a very long text containing multiple
email addresses like user1@example.com and another.user@domain.org, also
some text with numbers 12345, 67890. And one more email here:
final.email@test.co.uk.";
        int iterations = 100_000;

        // Шаблон для email-адресов
        Pattern emailPattern = Pattern.compile("\\b[A-Z0-9._%+-]+@[A-Z0-
9.-]+\\.[A-Z]{2,6}\\b", Pattern.CASE_INSENSITIVE);

        long totalTime = 0;
        for (int i = 0; i < iterations; i++) {
            long start = System.nanoTime();
            Matcher matcher = emailPattern.matcher(longText);
            while (matcher.find()) {
                // Do nothing, just find
            }
            long end = System.nanoTime();
            totalTime += (end - start);
        }
        System.out.println("Average time per 'find all emails' operation: "
+ (totalTime / iterations / 1_000_000.0) + " ms");

        // Сравнение с более агрессивным TLD, например, {2,}
        Pattern emailPatternOptimized = Pattern.compile("\\b[A-Z0-
9._%+-]+@[A-Z0-9.-]+\\.[A-Z]{2,}\\b", Pattern.CASE_INSENSITIVE);
        totalTime = 0;
        for (int i = 0; i < iterations; i++) {
            long start = System.nanoTime();
            Matcher matcher = emailPatternOptimized.matcher(longText);
            while (matcher.find()) {
                // Do nothing, just find
            }
            long end = System.nanoTime();
            totalTime += (end - start);
        }
        System.out.println("Average time per 'find all emails' (optimized
TLD): " + (totalTime / iterations / 1_000_000.0) + " ms");
    }
}
```

## 7.9. Бенчмаркинг различных подходов к сопоставлению

Бенчмаркинг — это систематическое сравнение производительности разных реализаций одной и той же задачи. Для Regex это может быть сравнение:

- Различных вариантов одного и того же регулярного выражения.
- Regex по сравнению со стандартными строковыми методами.
- Различных флагов компиляции.
- Использование `Matcher.find()` vs `Matcher.matches()` (когда применимо).

**Принципы бенчмаркинга**:

- **Повторяемость**: Выполняйте операции многократно (например, миллионы раз) для получения стабильных средних значений.
- **Разогрев JVM**: Перед началом измерений выполняйте несколько “разогревочных” итераций, чтобы позволить JIT-компилятору оптимизировать код.
- **Изоляция**: Измеряйте только интересующий вас код. Избегайте влияния операций ввода-вывода, создания объектов, которые не относятся к Regex, и т.д.
- **Используйте `System.nanoTime()`**: Для точных измерений времени.
- **Библиотеки для бенчмаркинга**: Для серьезного бенчмаркинга используйте Java Microbenchmark Harness (JMH), который учитывает нюансы JVM (оптимизации, сборка мусора).

**Пример (Java) - концепт JMH**:

```
// Пример с JMH (требует настройки JMH проекта)
/*
import org.openjdk.jmh.annotations.*;
import java.util.regex.Pattern;
import java.util.concurrent.TimeUnit;

@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
@Fork(value = 1, jvmArgs = {"-Xms2G", "-Xmx2G"})
@Warmup(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 5, time = 1, timeUnit = TimeUnit.SECONDS)
public class EmailValidationBenchmark {

    @Param({"test@example.com",
"another.long.email.address@sub.domain.co.uk", "invalid-email"})
    public String email;

    private Pattern compiledPattern;

    @Setup
    public void setup() {
        compiledPattern = Pattern.compile("^[A-Z0-9._%+-]+@[A-Z0-9.-]+\\.[A-
Z]{2,6}\$", Pattern.CASE_INSENSITIVE);
    }

    @Benchmark
    public boolean testEfficientRegex() {
        return compiledPattern.matcher(email).matches();
    }
```

```
    @Benchmark
    public boolean testInefficientRegex() {
        return Pattern.compile("^[A-Z0-9._%+-]+@[A-Z0-9.-]+\\.[A-Z]{2,6}\$",
Pattern.CASE_INSENSITIVE).matcher(email).matches();
    }

    @Benchmark
    public boolean testStringContains() {
        return email.contains("@") && email.contains("."); // Очень простая,
неточная проверка
    }
}
*/
```

## 7.10. Советы по написанию эффективных и читаемых Regex-шаблонов

Хорошо написанное регулярное выражение должно быть не только эффективным, но и понятным.

1. **Читаемость превыше всего (поначалу)**: Сначала напишите шаблон, который работает и легко читается. Оптимизируйте только тогда, когда профилировщик указывает на Regex как на узкое место.
2. **Используйте `Pattern.COMMENTS` (`(?x)`)**: Для сложных шаблонов используйте этот флаг и форматируйте Regex с пробелами и комментариями. Это значительно улучшает читаемость.

```
// Пример из Главы 4.10 с Pattern.COMMENTS
Pattern ipPattern = Pattern.compile(
    "^" +                           // Start of the line
    "([01]?\\d\\d?|2[0-4]\\d|25[0-5])" + // First octet (0-255)
    "\\." +                         // Dot separator
    "([01]?\\d\\d?|2[0-4]\\d|25[0-5])" + // Second octet
    "\\." +
    "([01]?\\d\\d?|2[0-4]\\d|25[0-5])" + // Third octet
    "\\." +
    "([01]?\\d\\d?|2[0-4]\\d|25[0-5])" + // Fourth octet
    "\$",                            // End of the line
    Pattern.COMMENTS
);
// Примечание: комментарии # здесь не используются в Java строках,
только в самом regex.
// Если бы это был многострочный строковый литерал (Java 15+), можно
было бы так:
/*
Pattern formattedRegex = Pattern.compile("""
    ^                   # Start of the line
    (\\d{3})            # First three digits
    \\s*                # Optional space
    (\\d{3})            # Second three digits
```

```
    \\s*
    (\\d{4})\$            # Last four digits
    """, Pattern.COMMENTS);
*/
```

3. **Используйте незахватывающие группы** `(?:...)`: Если вам не нужны обратные ссылки, это немного улучшает производительность и делает намерение более ясным.
4. **Будьте точны со специальными символами**: Экранируйте символы, которые являются метасимволами (`.`, `*`, `+`, `?`, `(`, `)`, `[`, `]`, `{`, `}`, `|`, `^`, `\$`, `\`). Используйте `Pattern.quote()` для экранирования целых строк.

```
String literalString = "text.with*special+chars";
Pattern p = Pattern.compile(Pattern.quote(literalString)); //
Экранирует все специальные символы
```

5. **Тестируйте граничные случаи**: Проверяйте, как Regex ведет себя с пустыми строками, очень длинными строками, строками с неподходящими символами и т.д.
6. **Используйте онлайн-валидаторы**: Такие инструменты, как regex101.com, regexr.com, debuggex.com, позволяют визуализировать шаблон, пошагово отлаживать его и видеть, как он сопоставляет текст, а также объяснять его поведение.

Следуя этим советам, вы сможете писать не только мощные, но и поддерживаемые и высокопроизводительные регулярные выражения в ваших Java-приложениях.

---

🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥

Отлично! Вот восьмая глава, посвященная отладке и тестированию регулярных выражений:

---

# Глава 8: Отладка и Тестирование Регулярных Выражений

Регулярные выражения, несмотря на свою мощность, могут быть весьма сложными для написания, понимания и, что особенно важно, для отладки. Даже небольшая ошибка в шаблоне может привести к непредсказуемому поведению, пропуску совпадений или, наоборот, к захвату лишнего текста. Эта глава посвящена методам и инструментам, которые помогут вам писать надежные и корректные регулярные выражения.

## 8.1. Типичные ошибки при написании Regex

Понимание распространенных ошибок может помочь их избежать:

1. **Неэкранированные метасимволы**: Забыли экранировать специальные символы (например, `. + * ? ( ) [ ] { } | ^ \$ \`) когда вы хотите сопоставить их буквально. В Java это требует двойного экранирования (`\\.`, `\\+`).
   - *Ошибка*: `String regex = "c.t";` (совпадет с `cat`, `cbt`, `c!t`)
   - *Исправление*: `String regex = "c\\.t";` (совпадет только с `c.t`)
2. **Жадность/ленивость квантификаторов**: Неправильный выбор жадного (`*`, `+`) или ленивого (`*?`, `+?`) квантификатора приводит к захвату слишком большого или слишком малого текста.
   - *Ошибка*: HTML-парсинг `<img src=".*">` в `<img src="image1.png"> <img src="image2.gif">` (захватит всю строку между первым `<` и последним `>`).
   - *Исправление*: `<img src=".*?">` (захватит каждый тег по отдельности).
3. **Отсутствие якорей (^, \$ ) при валидации**: Забыли использовать `^` и `\$` при валидации всей строки, что позволяет совпадению находиться как подстрока.
   - *Ошибка*: `Pattern.matches("\\d{5}", "12345abc")` вернет `true` если `matches` был `find` или `lookingAt`

    - *Исправление*: `Pattern.matches("^\\d{5}\$", "12345abc")` вернет `false` (при `matches()`)
4. **Катастрофический откат**: Использование рекурсивных или перекрывающихся квантификаторов, как `(a+)+`, что приводит к экспоненциальному времени выполнения на определенных входных данных.
    - *Ошибка*: `(ab|a)*c` для строки `abababac`
    - *Исправление*: Замена на атомарные группы `(?>ab|a)*c` или переосмысление логики.
5. **Неправильное использование символьных классов**: Путаница между `\w` и `[a-zA-Z0-9_]`, или `\s` и (пробел).
6. **Неправильные символьные наборы**: Забыли о том, что `^` внутри `[]` меняет смысл на “НЕ”. `[a-z]` для диапазона, `[^a-z]` для отрицания.
7. **Сложность Regex-строки в Java**: В Java-строках `\` также является специальным символом, поэтому `\` в Regex превращается в `\\` в Java-строке.
    - *Ошибка*: `Pattern.compile("\d+");` (будет ошибка компиляции Java-строки, а не Regex)
    - *Исправление*: `Pattern.compile("\\d+");`

## 8.2. Использование онлайн-валидаторов и визуализаторов Regex

Это одни из самых полезных инструментов для отладки регулярных выражений. Они позволяют вводить шаблон и тестовый текст, а затем визуализируют процесс сопоставления, подсвечивают совпадения и объясняют каждый шаг шаблона.

**Рекомендуемые инструменты**:

- **regex101.com**: Очень популярный и мощный инструмент. Позволяет выбрать разные “флейворы” Regex (Java, PCRE, Python и т.д.), подсвечивает синтаксис, объясняет каждый компонент Regex, показывает группы захвата и производительность отката.
- **regexr.com**: Еще один отличный интерактивный инструмент с полезными шпаргалками.
- **debuggex.com**: Визуализирует регулярные выражения в виде диаграммы конечного автомата, что помогает понять поток совпадений.

**Как использовать**:

1. Откройте выбранный онлайн-инструмент.
2. Вставьте свое регулярное выражение в поле “Regex”.
3. Вставьте свой тестовый текст (с примерами как корректных, так и некорректных данных) в поле “Test String”.
4. Выберите соответствующий “flavor” (Java).
5. Изучите объяснения и визуализацию. Посмотрите, что совпадает, а что нет, и почему.

## 8.3. Пошаговая отладка: анализ процесса сопоставления

Понимание того, как Regex-движок обрабатывает шаблон и текст, является ключом к отладке. Онлайн-валидаторы (особенно regex101.com) предоставляют эту функциональность пошагово.

**Что смотреть**:

- **Текущая позиция**: Где движок находится в строке.
- **Текущий элемент шаблона**: Какой элемент Regex он пытается сопоставить.
- **Захват**: Какие символы захватываются какой группой.
- **Откат (Backtracking)**: Когда движок вынужден отступать, чтобы найти альтернативный путь совпадения. Чрезмерный откат — признак проблемы с производительностью.
- **Успех/неудача**: На каком шаге происходит успех или неудача сопоставления.

Пошаговый анализ позволяет определить, где именно шаблон расходится с вашими ожиданиями.

## 8.4. Написание юнит-тестов для регулярных выражений

Юнит-тесты — это основной способ гарантировать корректность ваших регулярных выражений. Для Java это означает использование фреймворков, таких как JUnit.

**Принципы тестирования Regex**:

- **Позитивные тесты**: Проверьте, что шаблон правильно совпадает с **корректными** входными данными.
- **Негативные тесты**: Проверьте, что шаблон **не совпадает** с **некорректными** входными данными (например, неверный формат email, слишком длинный номер).
- **Граничные случаи**: Проверьте пустые строки, строки, содержащие только разделители, максимальные и минимальные допустимые значения.
- **Извлечение групп**: Если шаблон захватывает группы, убедитесь, что извлекаемые значения соответствуют ожиданиям.

**Пример (Java с JUnit 5)**:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class EmailValidatorTest {

    // Скомпилированный шаблон, как в реальном приложении
    private static final Pattern EMAIL_PATTERN =
        Pattern.compile("^[\\w!#\$%&'*+/=?`{|}~^-]+(?:\\.[\\w!#\$%&'*+/=?
`{|}~^-]+)*@(?:[a-zA-Z0-9-]+\\.)+[a-zA-Z]{2,6}\$",
                        Pattern.CASE_INSENSITIVE);

    @Test
    void testValidEmails() {
        assertTrue(EMAIL_PATTERN.matcher("test@example.com").matches());

assertTrue(EMAIL_PATTERN.matcher("john.doe123@sub.domain.co.uk").matches());
        assertTrue(EMAIL_PATTERN.matcher("user@mail.info").matches()); //
Пример более длинного TLD
    }

    @Test
    void testInvalidEmails() {
        assertFalse(EMAIL_PATTERN.matcher("invalid-email").matches());
        assertFalse(EMAIL_PATTERN.matcher("user@domain").matches()); // Нет
TLD
        assertFalse(EMAIL_PATTERN.matcher("user@.com").matches());   //
Точка в начале домена

assertFalse(EMAIL_PATTERN.matcher("user@domain.toolongtld").matches()); //
TLD > 6 символов, если Pattern.compile("...{2,6}")

// Для Pattern.compile("...{2,}") этот тест может быть не актуален.
        assertFalse(EMAIL_PATTERN.matcher("").matches()); // Пустая строка
        assertFalse(EMAIL_PATTERN.matcher("user@domain..com").matches()); //
Две точки подряд в домене
    }
```

```
    @Test
    void testEmailPartsExtraction() {
        String email = "admin@mycompany.org";
        Matcher matcher = EMAIL_PATTERN.matcher(email);
        assertTrue(matcher.matches());
        // Проверяем, что группы захватываются правильно
        assertEquals("admin", matcher.group(1)); // Предполагая, что
username - группа 1
        assertEquals("mycompany.org", matcher.group(2)); // Предполагая, что
домен - группа 2
        // В нашем EMAIL_PATTERN нет захватывающих групп, кроме всей строки,
        // поэтому пример group(1) group(2) может быть неактуален без явного
добавления групп.
        // Для демонстрации: Pattern.compile("(\\w+)@(.*)")
    }
}
```

## 8.5. Тестирование граничных условий и некорректных входных данных

Это расширение негативных тестов. Граничные условия — это значения на краях допустимого диапазона (например, минимальная/максимальная длина, первое/последнее допустимое значение символа). Некорректные входные данные включают:

- **Пустые строки или строки из пробелов**.
- **Строки, содержащие только разделители**.
- **Слишком короткие/длинные строки**.
- **Символы, которые похожи, но не являются допустимыми** (например, другие виды пробельных символов, разные виды тире).
- **Строки, предназначенные для вызова катастрофического отката (ReDoS)**, если ваш Regex к этому предрасположен.

Тщательное тестирование этих случаев помогает сделать Regex более надежным.

## 8.6. Использование логгирования для отслеживания поведения Regex

В сложных случаях, когда онлайн-валидаторы недоступны или нужно отлаживать Regex в runtime внутри вашего приложения, логгирование может быть полезным.

Вы можете добавить логгирование для:

- Входной строки.
- Статуса `find()`, `matches()`, `lookingAt()`.
- Значений `start()`, `end()` и `group()` для каждого совпадения.
- Исключений `PatternSyntaxException`.

**Пример (Java)**:

```
import java.util.logging.Logger;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexLogger {
```

```
    private static final Logger LOGGER =
Logger.getLogger(RegexLogger.class.getName());
    private static final Pattern NUMBER_PATTERN = Pattern.compile("(\\d+)");

    public void processText(String text) {
        LOGGER.info("Processing text: '" + text + "'");
        Matcher matcher = NUMBER_PATTERN.matcher(text);

        while (matcher.find()) {
            LOGGER.info("  Found match: '" + matcher.group() +
                        "' (Start: " + matcher.start() + ", End: " +
matcher.end() + ")");
            LOGGER.info("  Captured group 1: '" + matcher.group(1) + "'");
        }
        if (!matcher.hitEnd()) {
            LOGGER.info("  No more matches found. End of input reached: " +
matcher.hitEnd());
        }
    }

    public static void main(String[] args) {
        RegexLogger logger = new RegexLogger();
        logger.processText("This text contains 123 numbers and 456 more.");
        logger.processText("No numbers here.");
    }
}
```

## 8.7. Разделение сложных Regex на более мелкие, управляемые части

Монолитные регулярные выражения могут быть кошмаром для чтения и отладки. Если ваш шаблон становится очень длинным и сложным, рассмотрите возможность его декомпозиции:

- **Используйте конкатенацию строк**: Создайте шаблон из нескольких строковых констант.
- **Используйте `Pattern.COMMENTS` (`(?x)`)**: Позволяет разбивать шаблон на несколько строк, добавлять пробелы и `#` комментарии для читаемости.
- **Разбивайте задачу**: Возможно, одну часть валидации можно сделать одним Regex, а другую — другим. Или использовать обычные строковые операции для предварительной фильтрации.

**Пример (Java)**: См. примеры в разделе 4.10 и 7.10 для построения сложных шаблонов с использованием строковой конкатенации и флага `Pattern.COMMENTS`.

## 8.8. Техники для улучшения читаемости Regex: комментарии и форматирование

Читаемость напрямую влияет на отлаживаемость. Если вы или кто-то другой не может быстро понять, что делает Regex, отладка становится очень трудной.

- **Комментарии (`(?#comment)`)**: Встраивайте короткие комментарии непосредственно в шаблон.
- **Флаг `Pattern.COMMENTS` (`(?x)`)**: Используйте его для многострочного форматирования, пробелов и `#` комментариев, чтобы сделать шаблон похожим на код.

- **Называйте группы**: Если ваш движок поддерживает именованные группы (`(?<name>...)`), используйте их. В Java это `(?<name>...)`. Это делает `group("name")` намного более читаемым, чем `group(1)`.

```
// Пример с именованными группами (Java поддерживает)
Pattern namedGroupPattern = Pattern.compile("^(?<year>\\d{4})-(?
<month>\\d{2})-(?<day>\\d{2})\$");
Matcher m = namedGroupPattern.matcher("2024-08-17");
if (m.matches()) {
    System.out.println("Year: " + m.group("year")); // 2024
    System.out.println("Month: " + m.group("month")); // 08
}
```

- **Используйте понятные имена переменных**: Если вы сохраняете шаблон в переменной, дайте ей осмысленное имя.

## 8.9. Автоматизированное тестирование Regex-шаблонов

Помимо простых юнит-тестов, рассмотрите более продвинутые подходы для автоматизации тестирования:

- **Наборы тестовых данных**: Создайте большие наборы корректных и некорректных примеров. Можно использовать генераторы данных или реальные (анонимизированные) данные.
- **Property-based testing**: Инструменты, такие как JUnit-Quickcheck (для Java), могут генерировать случайные входные данные на основе определенных свойств, что помогает найти граничные случаи, которые вы могли пропустить.
- **Интеграционные тесты**: Если Regex является частью большей системы, убедитесь, что он правильно интегрируется и работает в рамках полного процесса.

## 8.10. Методологии отладки сложных систем с Regex

В реальных приложениях Regex часто является лишь небольшой частью сложной логики.

1. **Изоляция**: Если подозреваете проблему с Regex, изолируйте его. Возьмите проблемную входную строку, сам шаблон и протестируйте их отдельно (например, в юнит-тесте или онлайн-валидаторе).
2. **Упрощение**: Если шаблон очень сложен, постепенно удаляйте его части, пока не найдете ту часть, которая вызывает проблему.
3. **Использование IDE-функций**: Многие IDE (IntelliJ IDEA, Eclipse) имеют встроенную поддержку для тестирования Regex. Вы можете ввести шаблон и тестовую строку прямо в IDE и увидеть результаты.
4. **Постепенное наращивание**: При создании нового сложного Regex, начинайте с простейшего рабочего варианта, а затем постепенно добавляйте сложности, тестируя каждый шаг.
5. **Ревью кода**: Включение Regex в ревью кода может помочь другим разработчикам найти потенциальные проблемы.

Эффективная отладка и тестирование регулярных выражений — это навык, который приходит с практикой и использованием правильных инструментов и методологий.

---

🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥

Конечно! Вот девятая глава, посвященная продвинутым темам и особенностям регулярных выражений в Java:

---

# Глава 9: Продвинутые Темы и Особенности Java Regex

Эта глава углубляется в менее распространенные, но важные аспекты работы с регулярными выражениями в Java, охватывая нюансы Unicode, взаимодействие с другими API и вопросы безопасности.

## 9.1. Поддержка Unicode в Java Regex: `\p{L}`, `\p{Nd}` и другие классы

Java `java.util.regex` полностью поддерживает Unicode. Это означает, что `\d`, `\w`, `\s` и другие стандартные классы символов могут сопоставлять символы Unicode, а не только ASCII. Кроме того, Java предоставляет специальные классы свойств Unicode.

- `\p{prop}`: Совпадает с любым символом, который обладает свойством `prop`.
- `\P{prop}`: Совпадает с любым символом, который НЕ обладает свойством `prop`.

Некоторые распространенные свойства Unicode:

- `\p{L}` или `\p{Letter}`: Любая буква (включая символы из разных алфавитов, таких как кириллица, греческий, арабский и т.д.).
  - `\p{Lu}`: Заглавная буква.
  - `\p{Ll}`: Строчная буква.
- `\p{N}` или `\p{Number}`: Любой числовой символ.
  - `\p{Nd}`: Десятичная цифра (эквивалентно `\d`).
- `\p{P}` или `\p{Punctuation}`: Любой знак препинания.
- `\p{Z}` или `\p{Separator}`: Любой пробельный символ, который не является управляющим символом.
- `\p{Sc}`: Символы валют ($, €, ¥).
- `\p{InCJKUnifiedIdeographs}`: Символы CJK (китайские, японские, корейские иероглифы).
- `\p{Block=Name}`: Символы из определенного блока Unicode (например, `\p{Block=Cyrillic}`).

Вы также можете использовать флаг `Pattern.UNICODE_CHARACTER_CLASS` (или `(?U)`), который придает `\d`, `\s`, `\w` и другим сокращенным классам Unicode-специфичное поведение. По умолчанию эти классы уже Unicode-aware в Java, но этот флаг может влиять на их поведение в соответствии с более строгими определениями Unicode (особенно полезно при работе с определенными категориями пробелов или идентификаторов).

**Пример (Java)**:

```java
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class UnicodeRegexExample {
    public static void main(String[] args) {
        String text = "Hello, мир! 123 Zahlen. Тест. こんにちは。";

        // \p{L}: любая буква (Letter)
        Pattern p1 = Pattern.compile("\\p{L}+");
        Matcher m1 = p1.matcher(text);
        System.out.println("All words (Unicode letters):");
        while (m1.find()) {
            System.out.println("  " + m1.group());
            // Output: Hello, мир, Zahlen, Тест, こんにちは
        }

        // \p{Nd}: любая десятичная цифра (Number, Decimal Digit)
```

```
        Pattern p2 = Pattern.compile("\\p{Nd}+");
        Matcher m2 = p2.matcher(text);
        System.out.println("\nAll numbers (Decimal Digits):");
        while (m2.find()) {
            System.out.println("  " + m2.group());
            // Output: 123
        }

        // \p{P}: любой знак препинания (Punctuation)
        Pattern p3 = Pattern.compile("\\p{P}");
        Matcher m3 = p3.matcher(text);
        System.out.println("\nAll punctuation:");
        while (m3.find()) {
            System.out.println("  " + m3.group());
            // Output: ,, .
        }

        // Пример с флагом UNICODE_CASE (для регистронезависимого Unicode
сопоставления)
        String turkishText = "İstanbul"; // Турецкое 'İ' (заглавное I с
точкой)
        Pattern p4 = Pattern.compile("istanbul", Pattern.CASE_INSENSITIVE |
Pattern.UNICODE_CASE);
        Matcher m4 = p4.matcher(turkishText);
        System.out.println("\nCase-insensitive Unicode match for 'istanbul'
in 'İstanbul': " + m4.find()); // true
    }
}
```

## 9.2. Работа с различными кодировками символов

В Java строки внутренне представлены в кодировке UTF-16. Это означает, что регулярные выражения в Java всегда работают с UTF-16 символами (code units).

- **Code Points vs. Code Units**: Символы Unicode могут быть представлены одним (Basic Multilingual Plane, BMP) или двумя (суррогатные пары для дополнительных символов) UTF-16 code units.
  - Regex-движок Java обычно обрабатывает code units. Это может быть проблемой для символов, представленных суррогатными парами. Например, `.` будет соответствовать только одному code unit, а не всему символу.
  - Для большинства повседневных задач это не проблема, но если вы работаете с редкими символами вне BMP, возможно, потребуется нормализация или более сложные Regex-шаблоны.
- **Чтение из файлов**: Убедитесь, что вы читаете текст из файлов с правильной кодировкой (например, `new InputStreamReader(fis, "UTF-8")`). Если входная строка уже прочитана в Java `String`, то ее исходная кодировка файла уже обработана.

## 9.3. Regex и интернационализация (i18n): проблемы и решения

Использование Regex в многоязычных приложениях требует осторожности.

- **Регистронезависимое сопоставление**: Флаг `Pattern.CASE_INSENSITIVE` (и `Pattern.UNICODE_CASE` для полной поддержки Unicode) необходим, так как правила сопоставления регистра различаются между языками.
- **Границы слов (`\b`)**: В некоторых языках (например, китайском, японском, тайском) нет явных разделителей слов. `\b` может работать некорректно или быть бесполезным. В таких случаях для токенизации лучше использовать специализированные библиотеки NLP.
- **Диакритические знаки и комбинируемые символы**: Один и тот же символ может быть представлен по-разному в Unicode (например, `é` может быть одним символом или `e` + комбинируемый акут). Для согласованного сопоставления рекомендуется нормализовать текст в одну форму (например, NFC или NFD) перед применением Regex. Класс `java.text.Normalizer` может помочь.
- **Правила сортировки (collation)**: Regex не предназначен для сопоставления на основе правил сортировки, специфичных для языка. Например, в испанском `ch` может рассматриваться как одна буква. Для таких задач лучше использовать `java.text.Collator`.

## 9.4. Особенности поведения Regex в разных версиях Java

`java.util.regex` постоянно развивается, и в новых версиях Java добавляются новые возможности или улучшается производительность.

- **Java 7**: Добавлены новые классы символов:
  - `\h`: горизонтальный пробел (например, пробел, табуляция).
  - `\v`: вертикальный пробел (например, новая строка, возврат каретки, новая страница).
  - `\R`: любая последовательность новой строки (совпадает с `\n`, `\r`, `\r\n`). Очень полезно для кросс-платформенного сопоставления строк.
- **Java 9**: Улучшена поддержка Unicode, включая новые свойства для классов символов, например, `\p{javaLowerCase}`, `\p{javaUpperCase}`.
- **Java 15+ (Text Blocks)**: Хотя это не Regex-функция, текстовые блоки (многострочные строковые литералы) значительно улучшают читаемость сложных регулярных выражений, особенно при использовании флага `Pattern.COMMENTS` (`(?x)`).

```
// Java 15+
Pattern complexPattern = Pattern.compile("""
    ^                   # Start of the line
    (?:                 # Non-capturing group for OR condition
      [a-z0-9_]+        # Username part (letters, digits, underscore)
    |                   # OR
      \\d{3}-\\d{2}-\\d{4} # Social Security Number format
    )
    \$                   # End of the line
    """, Pattern.CASE_INSENSITIVE | Pattern.COMMENTS);
```

## 9.5. Сравнение `java.util.regex` с другими Regex-движками

Движки регулярных выражений делятся на две основные категории по их реализации:

1. **На основе NFA (Nondeterministic Finite Automaton)**:
   - **Примеры**: Java (`java.util.regex`), Perl, Python (`re` module), Ruby.
   - **Характеристики**: Мощные. Поддерживают обратные ссылки, lookaheads/lookbehinds. Могут быть медленными (экспоненциальное время) в худших случаях (катастрофический откат) из-за алгоритма бэктрекинга (отката). Они находят *одно* совпадение, проверяя все возможные пути.
   - **Как они "думают"**: "Пытаюсь найти совпадение по этому пути. Если не получается, откатываюсь и пробую другой путь."
2. **На основе DFA (Deterministic Finite Automaton)**:

- **Примеры**: `awk`, `grep` (большинство версий), RE2 (Google's regex library).
- **Характеристики**: Гарантируют линейное время выполнения (быстрее). Не поддерживают некоторые продвинутые функции (обратные ссылки, условные выражения, некоторые lookaheads/lookbehinds).
- **Как они "думают"**: "Сканирую строку, и в любой точке я точно знаю, в каком состоянии нахожусь, и смогу ли я найти совпадение."

**Вывод**: Java-движок является NFA-движком. Это означает, что он очень мощный и гибкий, но вы должны быть осведомлены о проблеме катастрофического отката (см. Глава 7), чтобы избежать проблем с производительностью и безопасностью.

## 9.6. Создание пользовательских символьных классов

В Java вы можете создавать сложные символьные классы, используя операторы объединения, пересечения и вычитания внутри квадратных скобок `[]`.

- **Объединение (Union)**: Просто перечисляйте символы или диапазоны.
  - `[a-z[A-Z]]` (совпадает с любой латинской буквой, эквивалентно `[a-zA-Z]`).
  - `[0-9[\p{L}]]` (любая цифра или любая буква).
- **Пересечение (Intersection)**: Используется `&&`. Совпадает с символами, которые принадлежат обоим наборам.
  - `[a-z&&[^aeiou]]` (любая строчная согласная буква).
  - `[\\p{L}&&[^\\p{Lu}]]` (любая буква, кроме заглавной, т.е. строчная буква).
- **Вычитание (Subtraction)**: Комбинация отрицания `^` и `&&`.
  - `[\\p{ASCII}&&[^\\p{Digit}]]` (любой ASCII символ, который не является цифрой).

**Пример (Java)**:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class CustomCharClassExample {
    public static void main(String[] args) {
        String text = "abcdeFGHIJ12345!@#\$";

        // Совпадение со строчными буквами, которые не являются гласными
        // [a-z] - строчные буквы, [^aeiou] - не гласные
        Pattern p1 = Pattern.compile("[a-z&&[^aeiou]]+");
        Matcher m1 = p1.matcher(text);
        System.out.println("Consonants:");
        while (m1.find()) {
            System.out.println("  " + m1.group()); // bcd, fg, hj
        }

        // Совпадение с буквами или цифрами (объединение)
        Pattern p2 = Pattern.compile("[\\p{L}[0-9]]+");
        Matcher m2 = p2.matcher(text);
        System.out.println("\nLetters or Digits:");
        while (m2.find()) {
            System.out.println("  " + m2.group()); // abcdeFGHIJ12345
        }
    }
}
```

## 9.7. Использование Regex с потоками данных (Streams API)

В Java 8 и более поздних версиях вы можете интегрировать регулярные выражения с Streams API для более функционального стиля обработки текста.

- `Pattern.splitAsStream(CharSequence input)`: Возвращает `Stream<String>`, разбивая входную строку по шаблону. Это удобная альтернатива `Pattern.split()`.
- Использование `Matcher` внутри потоков: Хотя `Matcher` не потокобезопасен, вы можете создавать его внутри лямбда-выражений, чтобы обрабатывать каждый элемент потока.

**Пример (Java)**:

```
import java.util.regex.Pattern;
import java.util.stream.Collectors;
import java.util.List;

public class RegexStreamsExample {
    public static void main(String[] args) {
        String sentence = "apple,banana;orange grape\tkiwi";

        // Использование splitAsStream для разбиения и сбора в список
        Pattern delimiterPattern = Pattern.compile("[,;\\s\\t]+");
        List<String> words = delimiterPattern.splitAsStream(sentence)
                                             .filter(s -> !s.isEmpty()) //
Отфильтровать пустые строки, если есть
                                             .collect(Collectors.toList());

        System.out.println("Words from sentence (splitAsStream): " + words);
// [apple, banana, orange, grape, kiwi]

        // Использование find() с Stream.iterate() (более сложный сценарий
для итерации)
        String logData = "LogEntry1:ID=123; LogEntry2:ID=456;
LogEntry3:ID=789;";
        Pattern idPattern = Pattern.compile("ID=(\\d+)");

        List<String> ids = Pattern.compile("ID=(\\d+)")
                                  .matcher(logData)
                                  .results() // Java 9+ method: returns a
Stream<MatchResult>
                                  .map(matchResult -> matchResult.group(1))
                                  .collect(Collectors.toList());

        System.out.println("Extracted IDs (using Matcher.results()): " +
ids); // [123, 456, 789]
    }
}
```

## 9.8. Интеграция Regex с другими Java-библиотеками

Некоторые популярные сторонние библиотеки Java предоставляют утилиты, которые упрощают или дополняют работу с регулярными выражениями:

- **Apache Commons Lang**: Содержит класс `StringUtils` с удобными методами, которые могут использовать Regex (хотя часто это просто обертки над `java.util.regex`). Например, `StringUtils.containsPattern()`, `StringUtils.splitByRegex()`.

```
// Пример (требуется библиотека Apache Commons Lang)
// import org.apache.commons.lang3.StringUtils;
// boolean hasDigit = StringUtils.containsPattern("Hello123World",
"\\d"); // true
```

- **Google Guava**: Предоставляет `CharMatcher` для эффективной работы с наборами символов. `CharMatcher` не является Regex, но для простых задач (например, проверка, содержит ли строка только цифры) он может быть намного производительнее, чем Regex.

```
// Пример (требуется библиотека Google Guava)
// import com.google.common.base.CharMatcher;
// boolean containsOnlyDigits = CharMatcher.inRange('0',
'9').matchesAllOf("12345"); // true
// boolean containsLetters =
CharMatcher.javaLetter().matchesAnyOf("Hello"); // true
```

## 9.9. Динамическое создание регулярных выражений

Иногда шаблон регулярного выражения необходимо строить или изменять во время выполнения программы, основываясь на пользовательском вводе или конфигурации.

- **Конкатенация строк**: Самый простой способ.

```
String userKeyword = "Java";
// Экранируем пользовательский ввод, чтобы избежать
PatternSyntaxException или инъекций Regex
String safeKeyword = Pattern.quote(userKeyword);
Pattern dynamicPattern = Pattern.compile(".*" + safeKeyword + ".*",
Pattern.CASE_INSENSITIVE);
```

- **Построение шаблона из списка**:

```
List<String> keywords = Arrays.asList("apple", "banana", "orange");
// Создаем альтернативный шаблон: (apple|banana|orange)
String joinedKeywords = keywords.stream()
                                .map(Pattern::quote) // Экранируем
каждое ключевое слово
                                .collect(Collectors.joining("|", "
(?:", ")")); // Незахаывающая группа
Pattern p = Pattern.compile(joinedKeywords, Pattern.CASE_INSENSITIVE);
System.out.println(p.pattern()); // (?:apple|banana|orange)
```

- **Внимание к безопасности**: При динамическом создании шаблонов из пользовательского ввода **всегда** используйте `Pattern.quote()` для экранирования строковых литералов,

которые должны быть частью Regex, но не должны интерпретироваться как метасимволы. Это предотвращает `PatternSyntaxException` и Regex Denial of Service (ReDoS) атаки.

## 9.10. Regex и безопасность: предотвращение Regex Denial of Service (ReDoS) атак

ReDoS — это атака отказа в обслуживании, при которой злоумышленник подает на вход приложению специально сформированную строку, которая вызывает экспоненциальный рост времени обработки регулярного выражения, приводя к зависанию или сбою приложения. Это происходит из-за шаблонов, подверженных катастрофическому откату.

**Примеры уязвимых шаблонов (см. 7.2)**:

- Вложенные квантификаторы: `(a+)+`, `(ab)*c`
- Чередующиеся альтернативы: `(a|aa)*b`
- Квантификаторы на группах, которые могут совпадать с пустыми строками: `(a*)*`

**Как предотвратить ReDoS**:

1. **Избегайте уязвимых шаблонов**: Самый важный шаг. Тщательно анализируйте свои регулярные выражения, особенно те, которые обрабатывают неконтролируемый пользовательский ввод.
2. **Используйте атомарные группы `(?>...)` и сверхагрессивные квантификаторы `*+`, `++`, `?+`**: Они отключают откат и могут значительно снизить риск ReDoS для определенных типов шаблонов.
3. **Ограничение длины ввода**: Прежде чем передавать пользовательский ввод в Regex, проверьте его длину. Очень длинные строки могут усугубить проблему даже для менее уязвимых шаблонов.
4. **Таймаут (Timeout)**: Хотя `java.util.regex` не имеет встроенной поддержки таймаута для операций сопоставления, вы можете реализовать это на уровне приложения, выполняя Regex в отдельном потоке и прерывая его, если операция занимает слишком много времени.

```
// Концептуальный пример таймаута (требует более сложной реализации
потоков)
/*
ExecutorService executor = Executors.newSingleThreadExecutor();
Future<Boolean> future = executor.submit(() -> {
    Matcher m = pattern.matcher(userInput);
    return m.matches();
});
try {
    Boolean result = future.get(5, TimeUnit.SECONDS); // Таймаут 5
секунд
    // ...
} catch (TimeoutException e) {
    future.cancel(true); // Прервать выполнение
    System.err.println("Regex operation timed out!");
} finally {
    executor.shutdownNow();
}
*/
```

5. **Тестирование на ReDoS**: Используйте инструменты для анализа безопасности (например, `safe-regex` для Node.js, концептуально применимо к другим языкам), чтобы выявить

уязвимые шаблоны.

Понимание этих продвинутых тем позволит вам писать более надежные, производительные и безопасные решения с использованием регулярных выражений в Java.

---

🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥🟥

Отлично! Вот заключительная, десятая глава, посвященная расширенным сценариям использования и лучшим практикам применения регулярных выражений в Java:

---

# Глава 10: Расширенные Сценарии Использования и Лучшие Практики

Эта глава обобщает знания о регулярных выражениях, рассматривая их применение в более широком контексте, включая взаимодействие с другими технологиями, паттерны проектирования и взгляды на будущее.

## 10.1. Полный пример: Построение простого парсера данных

Рассмотрим более сложный пример извлечения нескольких полей данных из текстовой строки, имитирующей запись журнала или конфигурации.

**Сценарий**: Парсинг строки, представляющей собой запись о транзакции: `"TRANS_ID:XYZ123 | TIMESTAMP:2024-08-17 14:30:00 | AMOUNT:150.75 | STATUS:COMPLETED"`

Нам нужно извлечь `TRANS_ID`, `TIMESTAMP`, `AMOUNT` и `STATUS`.

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;
import java.util.HashMap;
import java.util.Map;
import java.util.Optional;

public class TransactionParser {

    // Regex с именованными группами для удобства доступа (Java 7+)
    private static final Pattern TRANSACTION_PATTERN = Pattern.compile(
        "TRANS_ID:(?<transId>\\w+) " +
        "\\| TIMESTAMP:(?<timestamp>\\d{4}-\\d{2}-\\d{2}
\\d{2}:\\d{2}:\\d{2}) " +
        "\\| AMOUNT:(?<amount>\\d+\\.\\d{2}) " +
        "\\| STATUS:(?<status>\\w+)\$"
    );

    private static final DateTimeFormatter DATE_TIME_FORMATTER =
        DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
```

```
    public static Optional<Map<String, String>> parseTransaction(String
logEntry) {
        Matcher matcher = TRANSACTION_PATTERN.matcher(logEntry);

        if (matcher.matches()) {
            Map<String, String> transactionData = new HashMap<>();
            transactionData.put("transId", matcher.group("transId"));
            transactionData.put("timestamp", matcher.group("timestamp"));
            transactionData.put("amount", matcher.group("amount"));
            transactionData.put("status", matcher.group("status"));
            return Optional.of(transactionData);
        }
        return Optional.empty();
    }

    // Метод для дальнейшей обработки и типизации данных
    public static class TransactionData {
        public String transId;
        public LocalDateTime timestamp;
        public double amount;
        public String status;

        @Override
        public String toString() {
            return "TransactionData{" +
                   "transId='" + transId + '\'' +
                   ", timestamp=" + timestamp +
                   ", amount=" + amount +
                   ", status='" + status + '\'' +
                   '}';
        }
    }

    public static Optional<TransactionData> parseAndTypeTransaction(String
logEntry) {
        Matcher matcher = TRANSACTION_PATTERN.matcher(logEntry);

        if (matcher.matches()) {
            TransactionData data = new TransactionData();
            data.transId = matcher.group("transId");

            try {
                data.timestamp =
LocalDateTime.parse(matcher.group("timestamp"), DATE_TIME_FORMATTER);
                data.amount = Double.parseDouble(matcher.group("amount"));
            } catch (DateTimeParseException | NumberFormatException e) {
                System.err.println("Error parsing data types: " +
e.getMessage());
                return Optional.empty();
```

```
            }
            data.status = matcher.group("status");
            return Optional.of(data);
        }
        return Optional.empty();
    }

    public static void main(String[] args) {
        String entry1 = "TRANS_ID:XYZ123 | TIMESTAMP:2024-08-17 14:30:00 |
AMOUNT:150.75 | STATUS:COMPLETED";
        String entry2 = "TRANS_ID:ABC789 | TIMESTAMP:2024-08-16 09:00:00 |
AMOUNT:25.00 | STATUS:PENDING";
        String entry3 = "INVALID_ENTRY";
        String entry4 = "TRANS_ID:DEF456 | TIMESTAMP:2024-08-17 15:00:00 |
AMOUNT:ABC.DE | STATUS:FAILED"; // Некорректный AMOUNT

        System.out.println("--- Parsing to Map ---");
        parseTransaction(entry1).ifPresent(System.out::println); //
{transId=XYZ123, status=COMPLETED, amount=150.75, timestamp=2024-08-17
14:30:00}
        parseTransaction(entry3).ifPresentOrElse(System.out::println, () ->
System.out.println("Entry 3 is invalid.")); // Entry 3 is invalid.

        System.out.println("\n--- Parsing and Typing ---");
        parseAndTypeTransaction(entry1).ifPresent(System.out::println);
        parseAndTypeTransaction(entry2).ifPresent(System.out::println);
        parseAndTypeTransaction(entry3).ifPresentOrElse(System.out::println,
() -> System.out.println("Entry 3 is invalid."));
        parseAndTypeTransaction(entry4).ifPresentOrElse(System.out::println,
() -> System.out.println("Entry 4 is invalid due to parsing error."));
    }
}
```

## 10.2. Полный пример: Рефакторинг кода с использованием Regex

Регулярные выражения — мощный инструмент для поиска и замены в IDE, а также для программного рефакторинга (хотя для последнего чаще используются специализированные библиотеки парсинга кода).

**Сценарий в IDE**: Изменение имен полей Java из `_snake_case` в `camelCase` (например, `_user_name` в `userName`).

- **Поиск (Find)**: `_([a-z])`
- **Замена (Replace)**: `\$1` (или `\1` в некоторых IDE)
    - Пример: `_user_name` -> `user_name` (после первого запуска) -> `userName` (после второго запуска)
    - Часто IDE поддерживают более сложные замены с функцией `\L\l\$1` или `\U\u\$1` для изменения регистра.

**Программный рефакторинг (более сложный, но возможный пример)**: Предположим, вы хотите заменить все вызовы `System.out.println(someVar)` на `LOGGER.debug("someVar = {}", someVar);`

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class CodeRefactorer {

    // Ищем System.out.println(любая_переменная);
    private static final Pattern SYSOUT_PATTERN =

Pattern.compile("System\\.out\\.println\LEFTPAREN(.*?)\RIGHTPAREN;");

    public static String refactorSyso(String code) {
        Matcher matcher = SYSOUT_PATTERN.matcher(code);
        StringBuffer refactoredCode = new StringBuffer();

        while (matcher.find()) {
            String variableName = matcher.group(1).trim(); // Извлекаем имя
переменной
            // Заменяем на LOGGER.debug с использованием имя_переменной и {}
для форматирования
            // Экранируем \$ и { } в строке замены, если они должны быть
буквальными
            matcher.appendReplacement(refactoredCode, "LOGGER.debug(\"" +
variableName + " = {{}}\", " + variableName + ");");
        }
        matcher.appendTail(refactoredCode);
        return refactoredCode.toString();
    }

    public static void main(String[] args) {
        String originalCode = """
            public class MyClass {
                private String _user_name;

                public void doSomething() {
                    String message = "Hello";
                    System.out.println(message);
                    int count = 10;
                    System.out.println(count);
                }
            }
            """;

        System.out.println("Original Code:\n" + originalCode);
        String refactored = refactorSyso(originalCode);
        System.out.println("\nRefactored Code:\n" + refactored);
    }
}
```

**Важно**: Для сложного рефакторинга кода **строго рекомендуется** использовать специализированные библиотеки (например, JavaParser, Spoon, ANTLR) для построения абстрактных синтаксических деревьев (AST), так как Regex не может правильно обрабатывать вложенные структуры и контекст.

## 10.3. Регулярные выражения в работе с базами данных (например, PostgreSQL ~)

Многие системы баз данных поддерживают регулярные выражения в своих SQL-запросах для расширенного поиска строк. Синтаксис Regex может немного отличаться от Java, но основные концепции остаются теми же.

**PostgreSQL**:

- Оператор `~`: Сопоставление с регулярным выражением (регистрозависимое).
- Оператор `~*`: Сопоставление с регулярным выражением (регистронезависимое).
- Оператор `!~`: Не сопоставляет (регистрозависимое).
- Оператор `!~*`: Не сопоставляет (регистронезависимое).

**Пример SQL (PostgreSQL)**:

```
-- Найти всех пользователей, чье имя содержит "john" (регистронезависимо)
SELECT * FROM users WHERE name ~* 'john';

-- Найти все email-адреса, заканчивающиеся на .com или .org
SELECT email FROM customers WHERE email ~ '\\.(com|org)\$';

-- Найти продукты с кодом, начинающимся на две буквы и 3 цифры
SELECT product_name FROM products WHERE product_code ~ '^[A-Z]{2}\d{3}';
```

**MySQL**: Использует `REGEXP` или `RLIKE` операторы. **Oracle**: Использует функции `REGEXP_LIKE`, `REGEXP_SUBSTR`, `REGEXP_REPLACE` и другие.

При работе с базами данных важно помнить о различиях в синтаксисе Regex и потенциальных проблемах с производительностью при использовании Regex на больших наборах данных (индексы обычно не используются для Regex-поиска).

## 10.4. Использование Regex в скриптах сборки (Ant, Maven, Gradle)

Системы сборки часто используют регулярные выражения для фильтрации файлов, обработки имен ресурсов или определения условий.

- **Ant**: Использует `regexp` и `fileset` для включения/исключения файлов, а также задачи `replace` или `replaceregexp` для изменения содержимого файлов.

  ```
  <!-- Ant example to replace all occurrences of "DEBUG" with "INFO" in a
  file -->
  <replaceregexp file="mylog.txt"
                 match="DEBUG"
                 replace="INFO"
                 flags="g"/> <!-- 'g' for global replacement -->
  ```

- **Maven**: Использует фильтры ресурсов с синтаксисом Ant-style, а также может использовать плагины, которые в свою очередь используют Regex.

- **Gradle**: Более гибкий, так как основан на Groovy (который имеет мощную встроенную поддержку Regex). Вы можете использовать Regex в скриптах для обработки строк, фильтрации задач и т.д.

```
// Gradle example (in build.gradle) to filter files
task findJavaFiles {
    doLast {
        fileTree(dir: 'src').include('**/*.java').each { file ->
            if (file.name =~ /^A.*\.java\$/) { // Groovy Regex match
operator
                println "Found Java file starting with 'A':
\${file.name}"
            }
        }
    }
}
```

## 10.5. Регулярные выражения в IDE (IntelliJ IDEA, Eclipse) для поиска и замены

Современные IDE предоставляют мощные интерфейсы для использования регулярных выражений при поиске и замене текста в коде.

- **IntelliJ IDEA**:
  - `Ctrl+F` (Find), `Ctrl+R` (Replace): Включите опцию “Regex” (обычно кнопка `.*`).
  - Поддерживает именованные группы (`(?<name>...)`) и обратные ссылки `\$name` или `\$1`.
  - Поддерживает режимы `CASE_SENSITIVE`, `MULTILINE`, `DOTALL`.
  - `Ctrl+Shift+F` (Find in Files), `Ctrl+Shift+R` (Replace in Files): Глобальный поиск/замена.
- **Eclipse**:
  - `Ctrl+F` (Find/Replace): Отметьте “Regular expression” внизу окна.
  - Использует синтаксис, близкий к Java, но обратные ссылки в замене обычно `\\$1`, `\\$2`.
- **VS Code**:
  - `Ctrl+F` (Find), `Ctrl+H` (Replace): Активируйте иконку `.*` для включения Regex.
  - Поддерживает обратные ссылки `\$1`, `\$2`.

Эти функции значительно ускоряют рефакторинг и массовые изменения в больших кодовых базах.

## 10.6. Design Patterns, включающие Regex

Хотя Regex сам по себе не является паттерном проектирования, он часто используется в реализации различных паттернов:

- **Strategy Pattern**: Regex может быть частью стратегии для валидации или парсинга. Разные Regex-шаблоны могут представлять разные стратегии валидации для различных типов входных данных.
- **Template Method Pattern**: Базовый класс может определить скелет алгоритма, где одним из шагов является применение Regex, а конкретные реализации подклассов могут предоставлять свои собственные Regex-шаблоны.
- **Decorator Pattern**: Можно обернуть `Matcher` или `Pattern` для добавления дополнительной логики (например, кэширования результатов Regex-операций).
- **Builder Pattern**: Сложные Regex-шаблоны могут быть построены с использованием билдера для улучшения читаемости и модульности, хотя в Java это обычно просто конкатенация строк.
- **Interpreter Pattern**: Регулярные выражения являются простым примером паттерна “Интерпретатор” для текстового языка.

## 10.7. Сравнение Regex с более мощными инструментами парсинга (ANTLR, JavaCC)

Регулярные выражения идеально подходят для работы с **регулярными языками** — языками, которые могут быть описаны конечными автоматами. Это хорошо для извлечения информации из *полуструктурированных* данных (логов, конфигов), валидации форматов.

Однако, Regex не подходит для парсинга **контекстно-свободных языков** (таких как языки программирования, HTML/XML с их вложенными тегами) или для построения сложных синтаксических деревьев. Для таких задач используются:

- **Генераторы парсеров (Parser Generators)**:
  - **ANTLR (ANother Tool for Language Recognition)**: Мощный генератор парсеров для Java (и других языков). Позволяет определить грамматику языка в формате EBNF (Extended Backus-Naur Form) и сгенерировать лексер (токенизатор) и парсер. Идеален для создания компиляторов, интерпретаторов, преобразователей кода.
  - **JavaCC (Java Compiler Compiler)**: Еще один популярный генератор парсеров для Java.
- **Библиотеки для работы с DOM/SAX/StAX**: Для парсинга XML и HTML.
  - **Jsoup**: Для HTML в Java, простая библиотека для парсинга, манипуляций и извлечения данных из HTML.
  - Встроенные `javax.xml.parsers` (DOM/SAX) и `javax.xml.stream` (StAX) API для XML.

**Когда использовать что**:

- **Regex**: Простое сопоставление, валидация формата, поиск и замена в линейных текстах, извлечение простых полей.
- **Parser Generators (ANTLR/JavaCC)**: Сложные грамматики, языки программирования, структурированные данные с вложенностью, где важен синтаксический анализ.
- **DOM/SAX/StAX/Jsoup**: Парсинг XML/HTML.

## 10.8. Инструменты командной строки для работы с Regex (`grep`, `sed`, `awk`)

Регулярные выражения были изначально популяризованы в Unix-утилитах, которые до сих пор широко используются для обработки текста в командной строке.

- `grep`: (Global Regular Expression Print) Ищет строки, соответствующие регулярному выражению, и выводит их.
  - `grep "Error" mylog.txt`
  - `grep -E "ERROR|WARN" mylog.txt` (для расширенных Regex)
  - `grep -r "pattern" .` (рекурсивный поиск)
- `sed`: (Stream Editor) Потоковый редактор, используется для неинтерактивного преобразования текста. Часто используется для поиска и замены.
  - `sed 's/old_text/new_text/g' file.txt` (замена всех `old_text` на `new_text`)
  - `sed -E 's/([0-9]{4})-([0-9]{2})-([0-9]{2})/\3.\2.\1/g' dates.txt` (переформатирование дат)
- `awk`: Мощный язык обработки текста. Может выполнять сложные операции над столбцами и строками, используя Regex для сопоставления.
  - `awk '/ERROR/ { print \$0 }' mylog.txt` (печать строк, содержащих “ERROR”)
  - `awk -F ',' '{ print \$2 }' data.csv` (печать второго поля CSV)

Эти инструменты демонстрируют универсальность и мощь Regex за пределами языков программирования.

## 10.9. Обзор сообщества и ресурсов по Regex

Сообщество регулярных выражений очень активно, и существует множество ресурсов для обучения и отладки:

- **Онлайн-валидаторы/визуализаторы**: regex101.com, regexr.com, debuggex.com (уже упоминались, но ключевые).
- **Книги**:
  - “Mastering Regular Expressions” by Jeffrey E.F. Friedl: Считается Библией по Regex, глубоко вдается в детали работы движков.
- **Веб-сайты**:
  - regular-expressions.info: Всеобъемлющий ресурс с подробными объяснениями синтаксиса, поведением движков и советами по оптимизации.
  - Stack Overflow: Большое количество вопросов и ответов по Regex.
- **Учебники и курсы**: Множество онлайн-курсов и интерактивных учебников, например, на Codecademy, freeCodeCamp.
- **Сообщества**: Форумы по программированию, группы в социальных сетях, где можно задать вопросы.

## 10.10. Перспективы развития регулярных выражений и их применения

Регулярные выражения продолжают развиваться, хотя базовый синтаксис остается стабильным.

- **Улучшенная поддержка Unicode**: С ростом использования Unicode, движки Regex будут продолжать улучшать свою поддержку для корректной обработки всех символов и свойств.
- **Производительность и безопасность**: Исследования в области Regex продолжаются, сосредоточившись на повышении производительности и снижении уязвимостей к ReDoS-атакам, возможно, за счет использования гибридных NFA/DFA подходов.
- **Интеграция с ML/AI**: Для задач обработки естественного языка (NLP) Regex остается полезным для предварительной обработки, токенизации и извлечения простых шаблонов. В то время как машинное обучение берет на себя более сложные семантические задачи, Regex может служить в качестве быстрой и эффективной “первой линии обороны” для структурированных паттернов.
- **Декларативные шаблоны**: Возможно, в будущем появятся более высокоуровневые, декларативные способы описания текстовых шаблонов, которые компилируются в оптимизированные Regex, что сделает их более доступными и менее подверженными ошибкам.
- **Применение в Big Data и Cloud**: В обработке больших данных и облачных вычислениях Regex будет по-прежнему использоваться для парсинга логов, фильтрации потоков данных и извлечения метрик из неструктурированных источников.

Регулярные выражения остаются одним из самых универсальных и мощных инструментов для работы с текстом в арсенале любого разработчика. Освоив их, вы значительно расширите свои возможности по манипулированию и анализу текстовых данных.

---

На этом завершается полное оглавление и содержание всех 10 глав по регулярным выражениям в Java. Надеюсь, это всеобъемлющее руководство будет вам полезно!