Регулярные выражение RegEx

На самом деле регулярное выражение – это шаблон для поиска строки в тексте. В Java исходным представлением этого шаблона всегда является строка, то есть объект класса `String`. Однако не любая строка может быть скомпилирована в регулярное выражение, а только та, которая соответствует правилам написания регулярного выражения – синтаксису, определенному в спецификации языка. Для написания регулярного выражения используются буквенные и цифровые символы, а также метасимволы – символы, имеющие специальное значение в синтаксисе регулярных выражений.
Например:

```
String regex=”java”; // шаблон строки ”java”;
String regex=”\\d{3}”; // шаблон строки из трех цифровых символов;
```

Создание регулярных выражений в Java

Чтобы создать регулярное выражение Java, нужно сделать два простых шага:

1. написать его в виде строки с учётом синтаксиса регулярных выражений;
2. скомпилировать эту строку в регулярное выражение;

Работа с регулярными выражениями в любой Java-программе начинается с создания объекта класса `Pattern`. Для этого необходимо вызвать один из двух имеющихся в классе статических методов `compile`. Первый метод принимает один аргумент – строковый литерал регулярного выражения, а второй – плюс еще параметр, включающий режим сравнения шаблона с текстом:

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

Список возможных значений параметра `flags` определен в классе `Pattern` и доступен нам как статические переменные класса. Например:

```
Pattern pattern = Pattern.compile("java", Pattern.CASE_INSENSITIVE);//поиск совпадений с шаблоном будет производиться без учета регистра символов.
```

По сути, класс `Pattern` — это конструктор регулярных выражений. Под «капотом» метод `compile` вызывает закрытый конструктор класса `Pattern` для создания скомпилированного представления. Такой способ создания экземпляра шаблона реализован с целью создания его в виде неизменяемого объекта. При создании производится синтаксическая проверка регулярного выражения. При наличии ошибок в строке – генерируется исключение `PatternSyntaxException`.

Синтаксис регулярных выражений

Синтаксис регулярных выражений основан на использовании символов `<([{\^-=$!|]})?*+.>`, которые можно комбинировать с буквенными символами. В зависимости от роли их можно разделить на несколько групп:

1. Метасимволы для поиска совпадений границ строк или текста

Метасимвол	Назначение
^	начало строки
$	конец строки
\b	граница слова
\B	не граница слова
\A	начало ввода

\G	конец предыдущего совпадения
\Z	конец ввода
\z	конец ввода

2. Метасимволы для поиска символьных классов

Метасимвол	**Назначение**
\d	цифровой символ
\D	нецифровой символ
\s	символ пробела
\S	непробельный символ
\w	буквенно-цифровой символ или знак подчёркивания
\W	любой символ, кроме буквенного, цифрового или знака подчёркивания
.	любой символ

3. Метасимволы для поиска символов редактирования текста

Метасимвол	**Назначение**
\t	символ табуляции
\n	символ новой строки
\r	символ возврата каретки
\f	переход на новую страницу
\u 0085	символ следующей строки
\u 2028	символ разделения строк
\u 2029	символ разделения абзацев

4. Метасимволы для группировки символов

Метасимвол	**Назначение**
[абв]	любой из перечисленных (а,б, или в)
[^абв]	любой, кроме перечисленных (не а,б, в)
[a-zA-Z]	слияние диапазонов (латинские символы от a до z без учета регистра)
[a-d[m-p]]	объединение символов (от a до d и от m до p)
[a-z&&[def]]	пересечение символов (символы d,e,f)
[a-z&&[^bc]]	вычитание символов (символы a, d-z)

5. Метасимволы для обозначения количества символов – квантификаторы. Квантификатор всегда следует после символа или группы символов.

Метасимвол	**Назначение**
?	один или отсутствует
*	ноль или более раз
+	один или более раз
{n}	n раз
{n,}	n раз и более
{n,m}	не менее n раз и не более m раз

Жадный режим квантификатора

Особенностью квантификаторов является возможность использования их в разных режимах: жадном, сверхжадном и ленивом. **Сверхжадный режим** включается добавлением символа «+» после квантификатора, а ленивый – символа «?». Например:

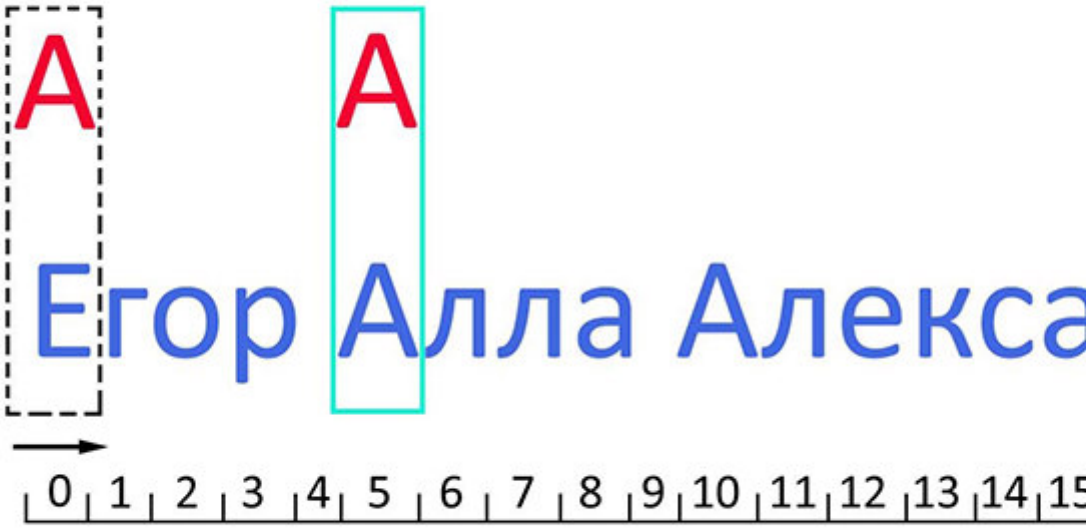
```
"A.+a" // жадный режим
"A.++a" // сверхжадный режим
"A.+?a" // ленивый режим
```

Попробуем на примере этого шаблона разобраться в работе квантификаторов в различных режимах. По умолчанию квантификатор работает в жадном режиме. Это означает, что он ищет максимально длинное совпадение в строке. В результате выполнения этого кода:

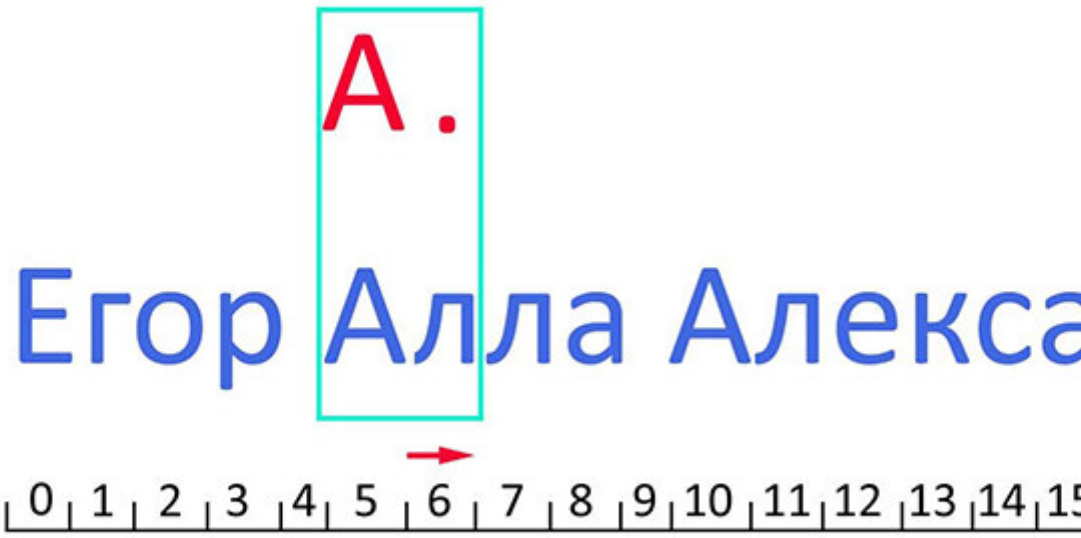
```
public static void main(String[] args) {
    String text = "Егор Алла Александр";
    Pattern pattern = Pattern.compile("A.+a");
    Matcher matcher = pattern.matcher(text);
    while (matcher.find()) {
        System.out.println(text.substring(matcher.start(), matcher.end()));
    }
}
```

мы получим такой вывод: ***Алла Алекса*** Алгоритм поиска по заданному шаблону "`A.+a`", выполняется в следующей последовательности:

1. В заданном шаблоне первый символ – это русский символ буквы `A`. `Matcher` сопоставляет его с каждым символом текста, начиная с нулевой позиции. На нулевой позиции в нашем тексте находиться символ `E`, поэтому `Matcher` перебирает последовательно символы в тексте, пока не встретит совпадение с шаблоном. В нашем примере это символ на позиции №5.

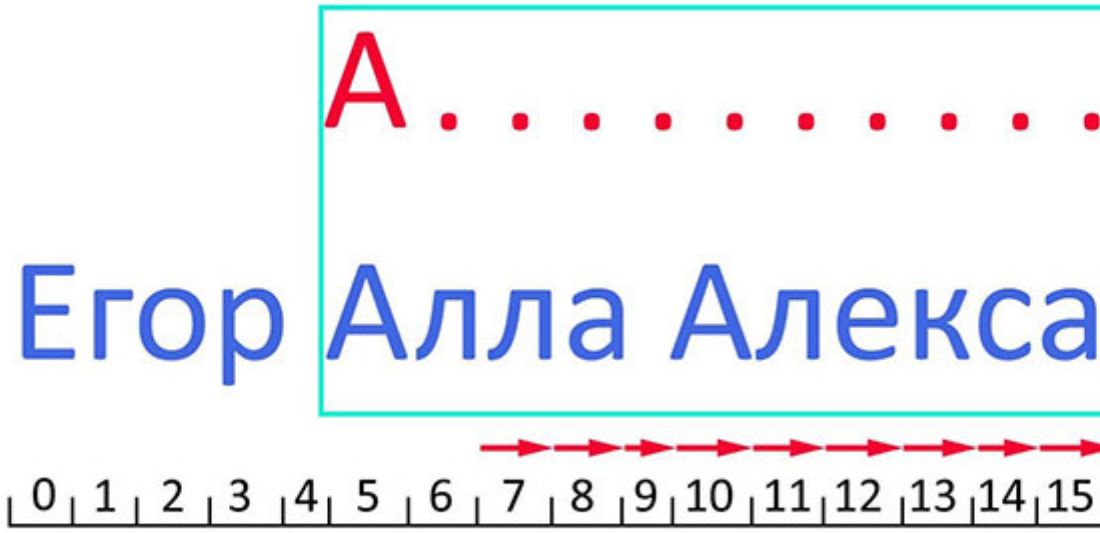


2. После того, как найдено совпадение с первым символом шаблона, `Matcher` сверяет соответствие со вторым символом шаблона. В нашем случае это символ «`.`», который обозначает любой символ.



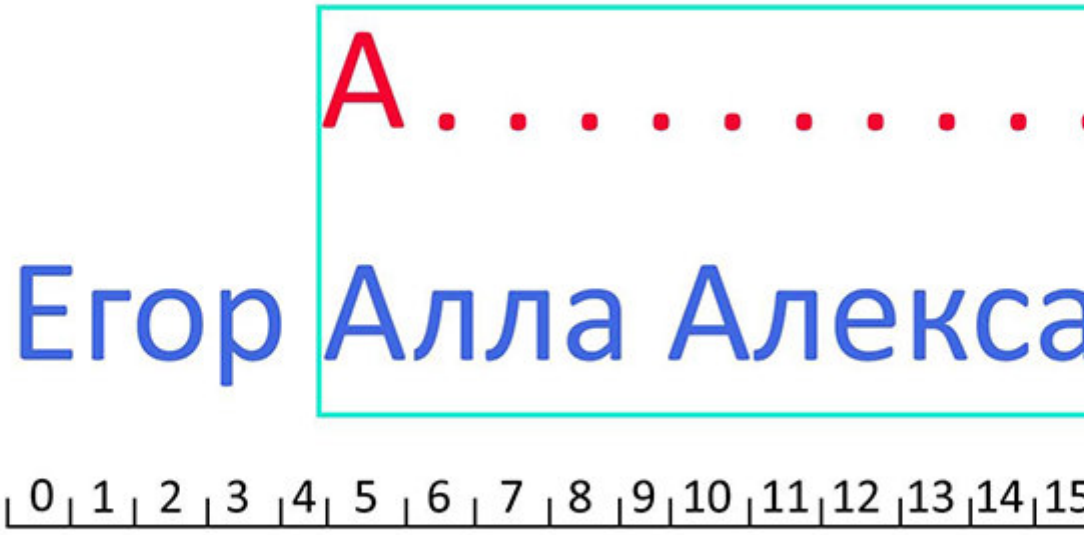
На шестой позиции – символ буквы `л`. Разумеется, он соответствует шаблону «любой символ».

3. `Matcher` переходит к проверке следующего символа из шаблона. В нашем шаблоне он задан с помощью квантификатора «`.+`». Поскольку количество повторений «любого символа» в шаблоне – один и более раз, `Matcher` берет по очереди следующий символ из строки и проверяет его на соответствие шаблону, до тех пор, пока будет выполняться условие «любой символ», в нашем примере – до конца строки (с поз. №7 -№18 текста).

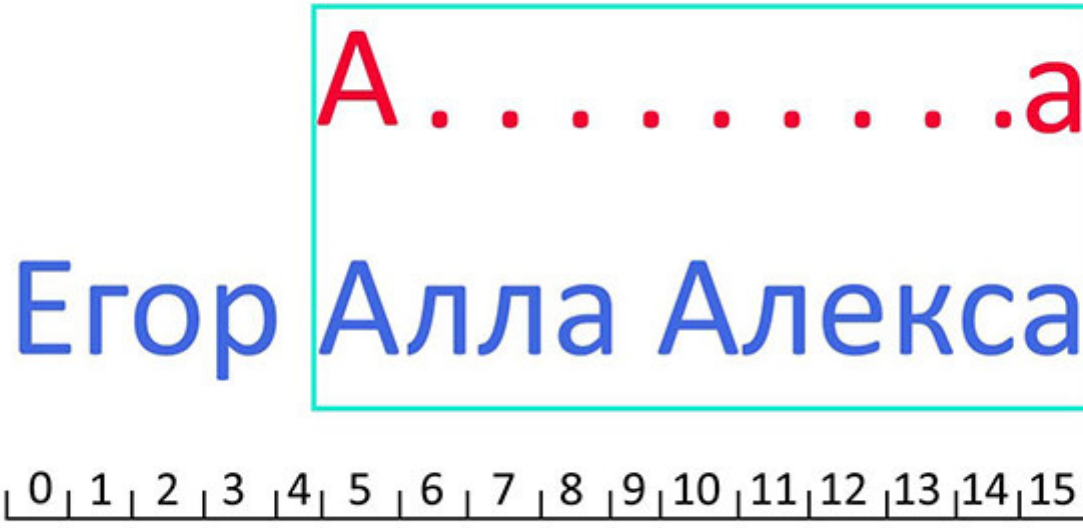


По сути, `Matcher`, захватывает все строку до конца – в этом как раз и проявляется его «жадность».

4. После того как `Matcher` дошел до конца текста и закончил проверку для части шаблона «`A.+`», Matcher начинает проверку для оставшейся части шаблона – символ буквы `a`. Так как текст в прямом направлении закончился, проверка происходит в обратном направлении, начиная с последнего символа:



5. `Matcher` «помнит» количество повторений в шаблоне «`.+`» при котором он дошел до конца текста, поэтому он уменьшает количество повторений на единицу и проверяет соответствие шаблона тексту, до тех пор пока не будет найдено совпадение:



Сверхжадный режим квантификатора

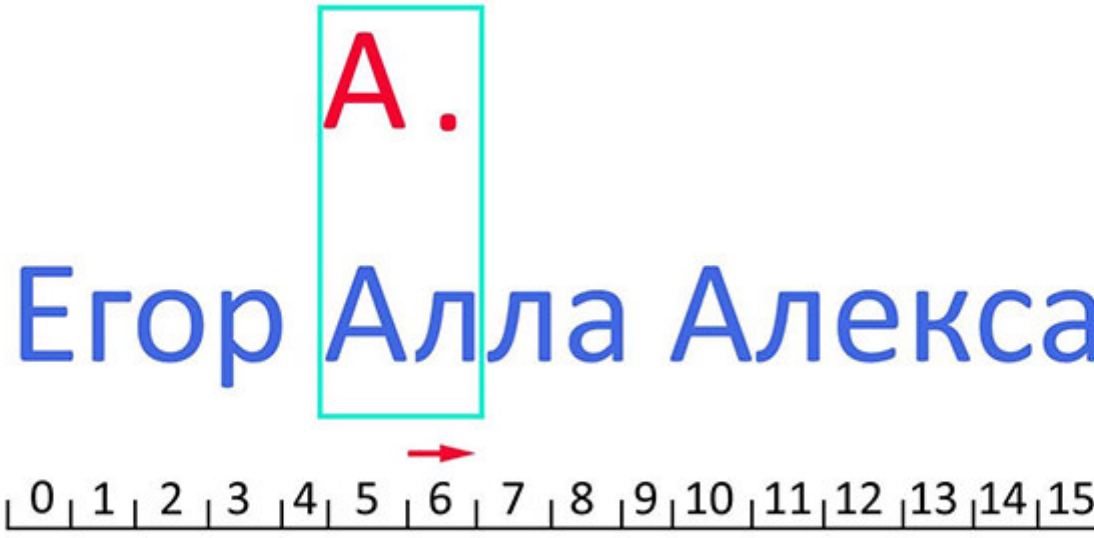
В сверхжадном режиме работа матчера аналогична механизму жадного режима. Отличие состоит в том, что при захватывании текста до конца строки поиск в обратном направлении не происходит. То есть первые три этапа при сверхжадном режиме будут аналогичны жадному режиму. После захвата всей строки матчер добавляет остаток шаблона и сравнивает с захваченной строкой. В нашем примере при выполнении метода

main с шаблоном "`A.++a`" совпадений не будет найдено.

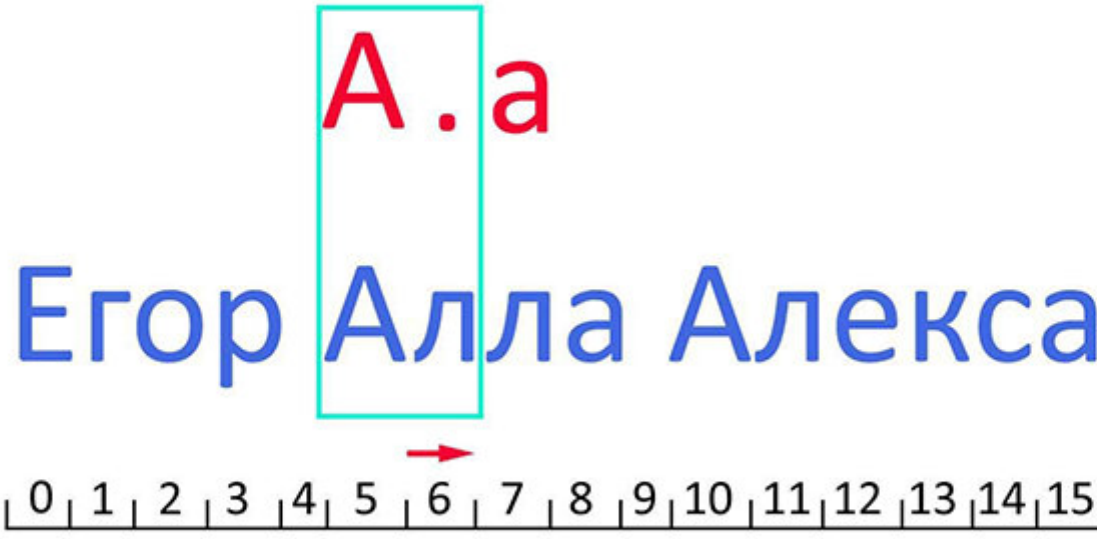
Ленивый режим квантификатора

1. В этом режиме на начальном этапе, как и в жадном режиме, ищется совпадение с первым символом шаблона:

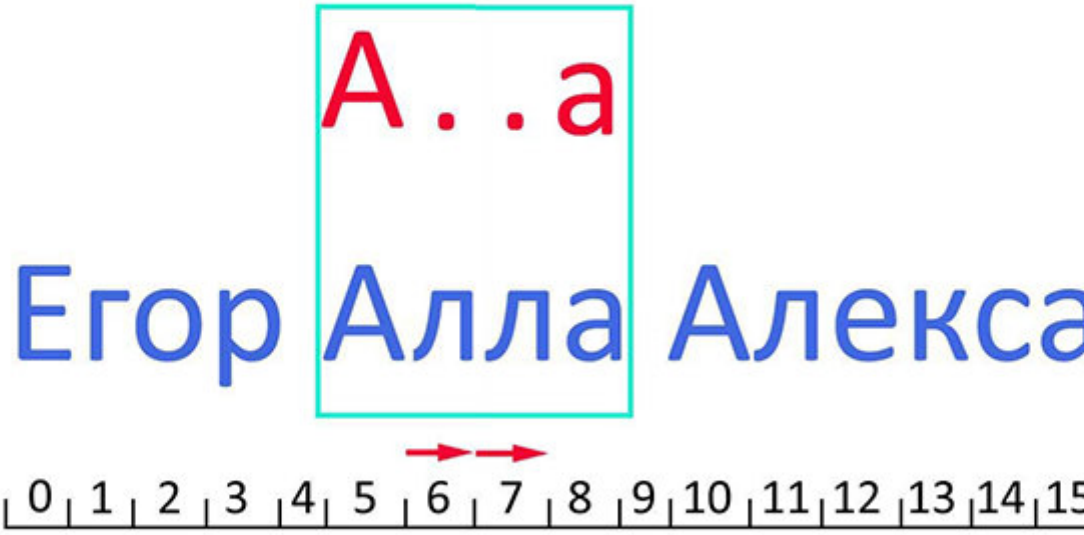
2. Далее ищется совпадение со следующим символом шаблона – любым символом:



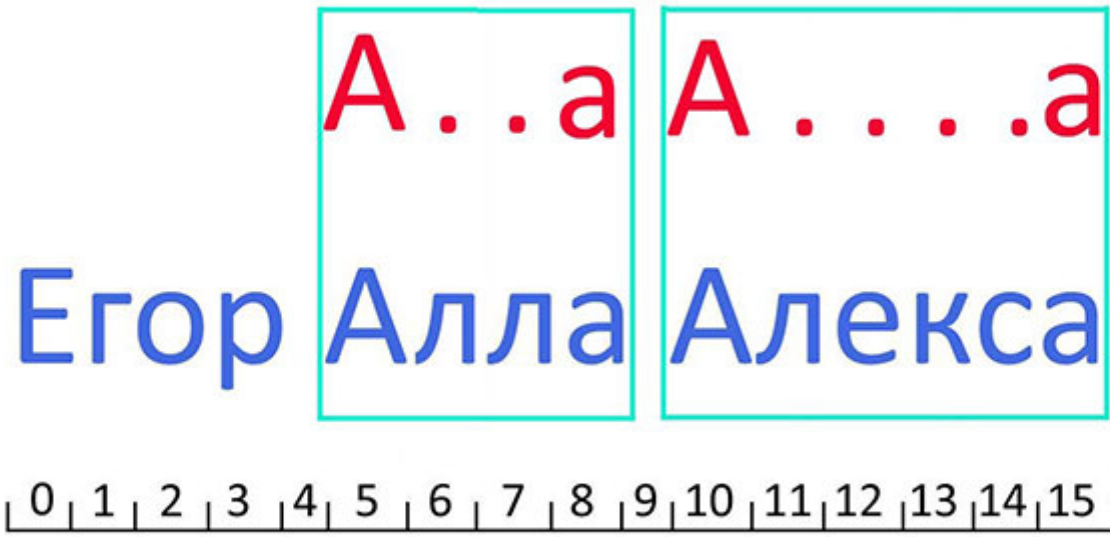
3. В отличие от жадного режима, в ленивом ищется самое короткое совпадение в тексте, поэтому после нахождения совпадения со вторым символом шаблона, который задан точкой и соответствует символу на позиции №6 текста, `Matcher` будет проверять соответствие текста остатку шаблона – символу «`a`»



4. Поскольку совпадение с шаблоном в тексте не найдено (на позиции №7 в тексте находится символ «`л`»), `Matcher` добавляет еще один «любой символ» в шаблоне, так как он задан как один и более раз, и опять сравнивает шаблон с текстом на позициях с №5 по №8:



5. В нашем случае найдено совпадение, но конец текста ещё не достигнут. Поэтому с позиции №9 проверка начинается с поиска первого символа шаблона по аналогичному алгоритму и далее повторяется вплоть до окончания текста.



В результате работы метода `main` при использовании шаблона "`A.+?a`" мы получим следующий результат: *Алла Алекса* Как видно из нашего примера, при использовании разных режимов квантификатора для одного и того же шаблона мы получили разные результаты. Поэтому необходимо учитывать эту особенность и выбирать нужный режим в зависимости от желаемого результата при поиске.

Экранирование символов в регулярных выражениях

Поскольку регулярное выражение в Java, а точнее — его исходное представление задается с помощью строкового литерала, необходимо учитывать те правила спецификации Java, которые касаются строковых литералов. В частности, символ обратной косой черты «\» в строковых литералах в исходном коде Java интерпретируется как символ управляющей последовательности, который предупреждает компилятор, что следующий за ним символ — специальный и что его нужно особым образом интерпретировать. Например:

```
String s="The root directory is \nWindows";//перенос Windows на новую строку
String s="The root directory is \u00A7Windows";//вставка символа параграфа
перед Windows
```

Поэтому в строковых литералах, которые описывают регулярное выражение, и используют символ «\» (например, для метасимволов) **его нужно удваивать**, чтобы компилятор байт-кода Java не интерпретировал его по-своему. Например:

```
String regex="\\s"; // шаблон для поиска символов пробела
String regex="\\"Windows\\""; // шаблон для поиска строки "Windows"
```

Двойной символ обратной косой черты также следует использовать для экранирования символов, задействованных в качестве специальных, если мы планируем их использовать как «обычные» символы. Например:

```
String regex="How\\?"; // шаблон для поиска строки "How?"
```

Методы класса Pattern

В классе `Pattern` есть и другие методы для работы с регулярными выражениями: **`String pattern()`** – возвращает исходное строковое представление регулярного выражения, из которого был создан объект `Pattern`:

```
Pattern pattern = Pattern.compile("abc");
System.out.println(Pattern.pattern())//"abc"
```

`static boolean matches(String regex, CharSequence input)` – позволяет проверить регулярное выражение, переданное в параметре regex на соответствие тексту, переданному в параметре `input`. Возвращает: *true* – если текст соответствует шаблону; *false* – в противном случае; Пример:

```
System.out.println(Pattern.matches("A.+a","Алла"));//true
System.out.println(Pattern.matches("A.+a","Егор Алла Александр"));//false
```

`int flags()` – возвращает значения параметра `flags` шаблона, которые были установлены при его создании, или 0, если этот параметр не был установлен. Пример:

```
Pattern pattern = Pattern.compile("abc");
System.out.println(pattern.flags());// 0
Pattern pattern = Pattern.compile("abc",Pattern.CASE_INSENSITIVE);
System.out.println(pattern.flags());// 2
```

`String[] split(CharSequence text, int limit)` – разбивает текст, переданный в качестве параметра на массив элементов `String`. Параметр `limit` определяет предельное количество совпадений, которое ищется в тексте:

- при `limit>0` – выполняется поиск `limit-1` совпадений;
- при `limit<0` – выполняется поиск всех совпадений в тексте
- при `limit=0` – выполняется поиск всех совпадений в тексте, при этом пустые строки в конце массива отбрасываются;

Пример:

```
public static void main(String[] args) {
    String text = "Егор Алла Анна";
    Pattern pattern = Pattern.compile("\\s");
    String[] strings = pattern.split(text,2);
    for (String s : strings) {
        System.out.println(s);
    }
    System.out.println("---------");
    String[] strings1 = pattern.split(text);
    for (String s : strings1) {
        System.out.println(s);
    }
}
```

Вывод на консоль: *Егор Алла Анна -------- Егор Алла Анна* Еще один метод класса для создания объекта `Matcher` рассмотрим ниже.

Методы класса Matcher

`Matcher` представляет собой класс, из которого создается объект для поиска совпадений по шаблону. `Matcher` – это «поисковик», «движок» регулярных выражений. Для поиска ему надо дать две вещи: шаблон поиска и «адрес», по которому искать. Для создания объекта `Matcher` предусмотрен следующий метод в классе `Pattern`: **`public Matcher matcher(CharSequence input)`** В качестве аргумента метод принимает последовательность символов, в котором будет производиться поиск. Это объекты классов, реализующих интерфейс `CharSequence`. в качестве аргумента можно передать не только `String`, но и `StringBuffer`, `StringBuilder`, `Segment` и `CharBuffer`. Шаблоном для поиска является объект класса `Pattern`, на котором вызывается метод `matcher`. Пример создания матчера:

```
Pattern p = Pattern.compile("a*b");// скомпилировали регулярное выражение в представление
Matcher m = p.matcher("aaaaab");//создали поисковик в тексте “aaaaab” по шаблону "a*b"
```

Теперь с помощью нашего «поисковика» мы можем искать совпадения, узнавать позицию совпадения в тексте, заменять текст с помощью методов класса. Метод **`boolean find()`** ищет очередное совпадение в тексте с шаблоном. С помощью этого метода и оператора цикла можно производить анализ всего текста по событийной модели (осуществлять необходимые операции при наступлении события – нахождении совпадения в тексте). Например, с помощью методов этого класса **`int start()`** и **`int end()`** можно определять позиции совпадения в тексте, а с помощью методов **`String replaceFirst(String replacement)`** и **`String replaceAll(String replacement)`** заменять в тексте совпадения на другой текст replacement. Пример:

```
public static void main(String[] args) {
    String text = "Егор Алла Анна";
    Pattern pattern = Pattern.compile("A.+?a");

    Matcher matcher = pattern.matcher(text);
    while (matcher.find()) {
        int start=matcher.start();
        int end=matcher.end();
        System.out.println("Найдено совпадение " + text.substring(start,end)
+ " с "+ start + " по " + (end-1) + " позицию");
    }
    System.out.println(matcher.replaceFirst("Ира"));
    System.out.println(matcher.replaceAll("Ольга"));
    System.out.println(text);
}
```

Вывод программы: *Найдено совпадение Алла с 5 по 8 позицию Найдено совпадение Анна с 10 по 13 позицию Егор Ира Анна Егор Ольга Ольга Егор Алла Анна* Из примера видно, что методы `replaceFirst` и `replaceAll` создают новый объект `String` – строку, представляющую собой исходный текст, в котором совпадения с шаблоном заменены на текст, переданный методу в качестве аргумента. Причём метод `replaceFirst` заменяет только первое совпадение, а `replaceAll` – все совпадения в тесте. Исходный текст остается без изменений. Использование других методов класса `Matcher`, а также примеры регулярных выражений можно посмотреть в этом цикле статей. Наиболее частые операции с регулярными выражениями при работе с текстом из классов `Pattern` и `Matcher` встроены в класс `String`. Это такие методы как `split`, `matches`, `replaceFirst`, `replaceAll`. Но на самом деле «под капотом» они используют классы `Pattern` и `Matcher`. Поэтому, если вам нужно заменить текст или сравнить строки в программе без написания лишнего кода, используйте методы класса `String`. Если же вам нужны расширенные возможности – вспомните о классах `Pattern` и `Matcher`.