Рядки в Java

Для хранения и обработки строк в Java определены два класса:

String – для неизменяемых строк;
StringBuffer – для строк, которые могут меняться.

Оба класса расширяют класс **Object**. Они находятся в пакете **java.lang**, поэтому для их подключения оператор **import** не нужен.

Строковые литералы в Java заключаются в двойные апострофы

"abc" задает строковый литерал ***abc***.

Если внутри строкового литерала необходимо задать символ апострофа, он задается с помощью символов **\"**

"it\"s" задает строковый литерал ***it"s***.

Ініціалізація об'єкта класу String

Может выполняться:

- с помощью оператора присваивания переменной класса **String** значения строковой переменной или строкового литерала

```
String str = "Строка 1";
```

- при создании объекта класса **String** с помощью оператора **new** с использованием одного из следующих конструкторов:

- **String()** – создается пустая строка;
- **String (String original)** – содержимое строки **original** копируется в другую строку;
- **String (StringBuffer buffer)** – содержимое строки **buffer** класса **StringBuffer** преобразуется в строку класса **String**;
- **String(byte[] bytes)** – строка создается из байтового массива **bytes** с использованием кодировки на данном компьютере по умолчанию;

- **String(byte[] bytes, int offset, int length)** – строка создается из части массива байт, начинающейся с индекса **offset** и содержащей **length** байт;
- **String (char [] value)** – строка создается из массива value символов ***Unicode***;
- **String(char[] value, int offset, int length)** — строка создается из части массива символов ***Unicode***, начинающейся с индекса offset и содержащей length символов.

```
String str = " abc ";
String str1 = new String(str); //Строка str1 примет значение "abc"
```

```
char[] charArray =
{"0", "1", "2", "3", "4"};
String str2 = new String(charArray);          //Строка str2 примет
                          значение " 01234 "
String str3 = new String(charArray,1,3);     // Строка str2 примет
                          значение "123"
```

Длина строки может быть определена с помощью метода **public int length()**

Для строк можно использовать операцию **сцепления (конкатенация)** двух или более строк – **"+".**

```
int strLength ="Строка 1".length();     // Значение
                    strLength будет равно 8.
```

```
String S = "Первая" + " строка";  // Строка S получит
                значение: "Первая строка"
```

Операцию конкатенации используют при переносе длинной строки на другую строку.

Строки класса **String** можно изменять, но при каждом изменении длины строки создается новый экземпляр строки

Класс **StringBuffer** похож на класс String, но строки, созданные с помощью этого класса можно модифицировать.

При изменении строки класса **StringBuffer** программа не создает новый строковый объект, а работает непосредственно с исходной строкой, все методы оперируют непосредственно с буфером, содержащим строку.

Класс **StringBuffer** обычно используется, когда строку приходится часто модифицировать с изменением ее длины.

Размещение строк в объекте StringBuffer :

- для объекта **StringBuffer** задается ***размер*** или ***емкость*** (*capacity*) *буферной* памяти для строки;
- строка символов в объекте **StringBuffer**, характеризуется также своей ***длиной***, которая может быть меньше или равна емкости буфера;
- если ***длина строки меньше емкости буфера***, то оставшаяся длина строки заполняется символом ***Unicode*** "\u0000";
- если в результате модификации строки ее ***длина станет больше емкости буфера***, емкость буфера автоматически увеличивается.

В классе StringBuffer имеется *три конструктора*:

```
StringBuffer();
StringBuffer(int length);
StringBuffer(String str).
```

Первый конструктор создает пустой объект **StringBuffer** с емкостью буферной памяти ***в 16 символов***.

Второй конструктор задает буфер с емкостью **length** для хранения строки.

Третий конструктор создает объект **StringBuffer** из объекта **String** с емкостью буфера, равной длине строки класса **String**.

Длину строки в объекте **StringBuffer** можно так же, как и для строки класса **String**, получить с помощью метода

```
public int length()
```

Текущую емкость буферной области можно получить с помощью метода

```
public int capacity()
```

Емкость буферной памяти можно также установить с помощью метода

```
public void ensureCapacity(int minimumCapacity)
```

Пример:

```
StringBuffer str = new StringBuffer("String buffer");
str.ensureCapacity(512);
```

Длина буферной памяти для строки str будет установлена равной 512 байтам.

Длина строки StringBuffer устанавливается с помощью метода **public void setLength(int newLength)**

```
StringBuffer str = new StringBuffer("String buffer");
str.setLength(40) ;
```

Если **новая длина больше старой**, увеличиваются длины строки и буфера, а новые символы заполняются нулями.

Если **новая длина меньше старой**, символы в конце строки отбрасываются, а размер буфера не изменяется.

Для преобразования строки StringBuffer в строку String используется метод **public String toString()**

Порівняння рядків

Поскольку в Java строки являются объектами, для сравнения строк можно использовать оператор "=="

Использование оператора "==" для сравнения строк может привести к неверному результату, если сравниваемые строки – разные объекты, поэтому более предпочтительным является использование метода **equals()**

```
public boolean equals(Object anObject)
```

Метод сравнивает строку, для которой вызывается метод, с объектом **anObject**. Результат вызова метода будет **true**, только если **anObject** является строкой и значения сравниваемых строк равны.

```
String str1 = new String("Строка");
String str2 = new String("Строка");
String str3 = str2 + "1";
int x=0, y=0;
if (str1.equals(str2))     x = 1;
if (str1.equals(str3))     y = 1;
```

В результате выполнения операторов переменная x получит значение 1, а значение y останется равным 0.

Для сравнения строк класса String определены методы (public):

- **boolean equalsIgnoreCase(String anotherString)** – сравнение значений строк без учета регистра букв;
- **boolean startsWith(String prefix)** – проверка, содержится ли строка prefix в начале проверяемой строки;
- **boolean startsWith(String prefix, int toffset)** – проверка, содержится ли подстрока строки prefix, начиная с позиции toffset в начале проверяемой строки;
- **boolean endsWith(String suffix)** – проверка, содержится ли строка prefix в конце проверяемой строки;
- **boolean regionMatches(int toffset, String other, int ooffset, int len)** – сравнивает len символов в двух строках, причем в первой строке сравниваемые символы начинаются с позиции **toffset** , а во второй строке other – с позиции **ooffset**;

Для сравнения строк класса String определены методы (public):

- **boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)** – выполняет ту же операцию, что и предыдущий метод, но если значение **ignoreCase** при вызове метода равно **true**, то регистры букв в сравниваемых подстроках игнорируются.

- **int compareTo(String anotherString)** – лексикографически сравнивает две строки и возвращает значение:

  - **0**, если строки равны по длине и имеют одинаковое значение;
  - **меньшее 0**, если в первой позиции, в которой символы строк не равны, код символа в первой строке меньше кода символа во второй строке или длина первой строки меньше длины второй строки все символы первой строки равны символам в тех же позициях второй строки;
  - **большее 0**, если в первой позиции, в которой символы строк не равны, код символа в первой строке больше кода символа во второй строке или длина первой строки больше длины второй строки.

Пример

```
String str1 = "abc";
String str2 = "aBc";
boolean comp12 = str1.equalsIgnoreCase(str2);
```

переменная comp12 получит значение true.

```
String str3 = "bcde";
String str4 = "abc";
boolean comp34 = str3.startsWith(str4,1);
```

переменная comp34 получит значение false

```
String str5 = "abc";
String str6 = "abcde";
int comp56 = str5.compareTo(str6);
```

переменная comp56 получит значение, меньшее 0

```
String str7 = new String("Строка1");
String str8 = new String("Новая строка");
int x;
if (str7.regionMatches(true, 0, str8, 6, 6))
x = 1;
```

Значение выражения в скобках будет равно true, поскольку первые 6 символов строки str7 совпадают с шестью символами строки str8, начиная с индекса 6 без учета регистра символов.

Пошук в рядках

Для поиска символов или последовательностей символов (только в строках класса **String**) используются следующие перегружаемые **методы indexOf()** (public):

- **int indexOf(int ch)** – возвращает первую позицию в строке, в которой встречается символ ch;
- **public int indexOf(int ch, int fromIndex)** – возвращает первую позицию в строке, начиная с позиции fromIndex,в которой встречается символ ch;
- **int indexOf(String str)** – возвращает первую позицию в строке, в которой встречается строка str;
- **public int indexOf(String str, int fromIndex)** – возвращает первую позицию в строке, начиная с позиции fromIndex,в которой встречается строка str.

Для каждого метода **indexOf()** имеется соответствующий метод **lastIndexOf()**, который выполняет поиск символа или строки не с начала, а с конца строки.

Если символ или строка не найдены в строке, в которой производится поиск, методы **indexOf()** и **lastIndexOf()** возвращают значение **-1.**

Пример

```
String str1 = new String("Строка1");
String str2 = new String("Новая строка");
int x, y, z;
x = str1.indexOf('к', 1);                    // x = 4
y = str2.indexOf("строка");                  // y = 6
z = str2.indexOf("строка1");                 // z = -1
```

Вилучення з рядків

Извлечение символов и подстрок из строк String и **StringBuffer** выполняется с помощью следующих методов (public):

- **char charAt(int index)** – возвращает значение символа строки в позиции index;
- **char[] toCharArray() –** возвращает массив символов – копию строки;
- **String substring(int beginIndex)** – возвращает строку, начинающуюся с позиции beginIndex исходной строки и до конца строки;
- **String substring(int beginIndex, int endIndex)** – возвращает строку, начинающуюся в позиции beginIndex и заканчивающуюся в позиции, на единицу меньшей endIndex в исходной строке;
- **void getChars (int srcBegin, int srcEnd, char[] dst, int dstBegin)** – копирует часть строки, начиная с символа в позиции **srcBegin** и заканчивая символом в позиции **dstBegin + (srcEnd - srcBegin) - 1** в символьный массив **dst**, начиная с позиции **dstBegin**.

Пример

```
String str1 = new String("Строка 1");
char firstSymbol = str1.charAt(0);
```

Переменная firstSymbolstr получит значение 'C'.

```
String str2 = new String("Новая строка 2");
String substr21 = str2.substring(6);
String substr22 = str2.substring(6,12);
```

Переменная substr21 получит значение "строка 2", а переменная substr22 – значение "строка".

```
StringBuffer str3 = new StringBuffer("String buffer");
char ch[] = new char[20];
str3.getChars(7, 10, ch, 0);
```

Извлечение символов строки str3 в символьный массив ch.

Модифікація рядків

Для модификации строк класса **String** определены следующие методы (public):

- **String concat(String str)** – возвращает исходную строку, в конец которой добавлена строка str;
- **String toLowerCase()** – возвращает строку, в которой все буквы переведены в нижний регистр;
- **String toUpperCase()** – возвращает строку, в которой все буквы переведены в верхний регистр;
- **String trim()** – возвращает строку, в которой удалены все пробельные символы (символы с кодами, не превышающими '\u0020') в начале и в конце строки;
- **String replace(char oldChar, char newChar)** – заменяет в строке все символы oldChar на newChar.

Пример

```
String str1 = new String("Строка 1");
String upperStr1 = str1.toUpperCase();
```

Переменная upperStr1 получит значение "СТРОКА 1".

```
String str2 = new String("  Строка 2  ");
str2 = str2.trim();
```

Переменная str2 получит значение "Строка2".

```
String str3 = new String("a:b:c:d");
str3 = str3. replace(':',',');
```

Переменная str3 получит значение "a,b,c,d".

Для создания строк из примитивных типов данных в классе **String** используются перегруженные статические методы **valueOf()**, в качестве аргумента которых задается константа, переменная или выражение примитивного типа (**boolean, char, int, long, float**, или **double**).

Возвращаемым значение метода является строка класса **String** – строковое представление аргумента.

```
String.valueOf(15);
```

возвращает строку "15"

- **static String valueOf(char[] data)** – возвращает строковое представление символьного массива data;

- **static String valueOf(char[] data, int offset, int count)** – возвращает строковое представление символьного массива data, начиная с позиции offset и размером count.

Методы класса StringBuffer могут непосредственно модифицировать строку

- **public void setCharAt(int index, char ch)** помещает символ **ch** в указанной позиции **index** строки.

```
StringBuffer str =
new StringBuffer("String buffer");
str.setCharAt(3, 'X');
```

Переменная str после выполнения метода setCharAt() будет содержать символы "StrXng buffer"

- **public void deleteCharAt(int index)** удаляет символ в заданной позиции **index** строки.

```
StringBuffer str = new
StringBuffer("String buffer");
str.deleteCharAt(2);
```

После удаления символа 'r' длина строки уменьшится на 1 и строка str будет иметь значение "Sting buffer".

• **public StringBuffer replace(int start, int end, String str)** заменяет подстроку в строке, начиная с символа в позиции **start** и до символа в позиции **end-1** строкой **str**.

```
StringBuffer str = new StringBuffer("String buffer");
str.replace(0,6,"Array");
```

После замены строка будет иметь вид "Array buffer" и длина строки уменьшится на 1.

•**public StringBuffer append (тип-параметра имя-параметра)** добавляет символы в конец строки

•**public StringBuffer insert(int offset, тип-параметра имя-параметра)** вставляет символы в любом месте строки, начиная с позиции offset.

Оба метода имеют несколько версий, позволяющих обрабатывать различные типы данных. Для append и insert **тип-параметра** может принимать значения:
Object, String, char[], boolean, char, int, long, float, double

Пример:

Добавление строкового представления целого числа в конец строки

```
StringBuffer str1 = new StringBuffer("String buffer");
int value = 15;
str1.append(value);                    // str1="String buffer15"
```

Вставка символов

```
StringBuffer str2 = new StringBuffer("String buffer");
int value = 15;
str.insert(6, value);                  // str2="String15 buffer"
```

- **public StringBuffer append(char[] str, int offset, int len)** - метод добавления
- **public StringBuffer insert(int offset0, char[] str, int offset1, int len)** - метод вставки

В качестве аргумента передается часть массива **char**, начиная с индекса **offset** и длиной **len**.

Регулярні вирази в Java

При работе с данными часто приходится выполнять операции поиска и замены по сложным алгоритмам. Для выполнения таких операций в языке Java используются **регулярные выражения**.

Регулярное выражение в языке Java является строкой.

Регулярные выражения используются для решения следующих задач:

- проверка данных на наличие некоторой последовательности данных, заданных с помощью определенного образца, называемого **шаблоном** (pattern);
- замена или удаление данных;
- извлечение некоторой последовательности из данных.

Синтаксис регулярного виразу

- алфавитно-цифровые символы, включая буквы кириллицы;
- символ '\\' – обратная косая черта (обратный слеш);
- символ '\0num' – восьмеричное число, где num – одна, две или три восьмеричные цифры;
- символ '\xhh' – код символа ASCII, где hh – две шестнадцатеричные цифры;
- символ '\uhhhh' – код символа Unicode, где hhhh – четыре шестнадцатеричные цифры;
- символ табуляции ('\t' или '\u0009');
- символ новой строки ('\n' или '\u000A');
- символ возврата каретки ('\r' или '\u000D');
- символ перехода к новой странице ('\f' или '\u000C');
- символ звукового сигнала ('\a' или '\u0007');
- символ **Escape** (**Esc**) ('\u001B');
- символ '\cx' – соответствует управляющему символу x (например, \cM соответствует символу **Ctrl**+**M** или символу возврата каретки).

Операція альтернації

В регулярных выражениях можно объединять несколько шаблонов, так чтобы найденная строка соответствовала хотя бы одному из них. Для этого служит **операция альтернации**, *которая в регулярных выражениях задается символом* "|".

Пример:

шаблон **"Имя | Фамилия"** означает поиск в исходной строке ."либо строки **"Имя"**, либо строки **"Фамилия**

Одиночний метасимвол

Метасимвол точка "**.**" внутри регулярного выражения точка соответствует любому одиночному символу, кроме символа перевода строки.

Пример:

шаблоне **"w.r"** соответствуют слова **war, world, forward** и т.д.

Квантіфікатори

Квантификаторы – это метасимволы, используемые для указания количественных отношений между символами в шаблоне и в искомой строке. Квантификатор может быть поставлен после одиночного символа или после группы символов.

Метасимвол "+" означает, что идущий перед ним символ соответствует нескольким идущим подряд таким символам в строке поиска. Количество символов может быть любым, но должен присутствовать хотя бы один символ.

Метасимвол "*" указывает, что идущий перед ним символ встречается нуль или более раз.

Метасимвол "?" указывает, что предшествующий ему символ должен встречаться либо один раз, либо не встречаться вообще.

Пример использования метасимвола "+"

"+wor"

Этому шаблону будут соответствовать слова **world** и **worry**, а слово **woman** соответствовать не будет.

Пример использования метасимвола "*"

"*wor"

.Этому шаблону будут соответствовать слова **world**, **worry** и **woman**

Пример использования метасимвола "?"

"?wor"

Этому шаблону будут соответствовать слова **world** и **woman**, а слово **worry** соответствовать не будет.

Если необходимо **указать точно количество повторений символа**, можно воспользоваться конструкцией

{n,m}

n – минимально допустимое количество повторений предшествующего символа, **m** – максимально допустимое количество повторений. Один из параметров **n** или **m** можно опустить.

Фактически квантификаторы "+", "*" и "**?**" являются частными случаями конструкции **{n,m}**: соответственно, **{1,}**, **{0,}** и **{0,1}**.

Пример:

"10{3,5}1" - 0 встречается как минимум 3 раза, но не более 5 раз.

"10{3,}1" - 0 встречается 3 или более раз.

"10{0,3}1" - 0 встречается не более 3 раз, но может вообще не встретиться.

"10{3}1" - 0 встречается ровно 3 раза.

В регулярных выражениях часто используют **сочетание метасимволов ".*"** . Ему соответствуют любые символы. По правилам обработки регулярных выражений находится самая длинная строка, все еще удовлетворяющая шаблону поиска.

Если необходимо ограничить поиск, следует после квантификатора (в том числе и символа "**?**") указать символ "**?**".

Пример:

Исходная строка:

"первый может стать как последний и последний может стать как первый."

Шаблон: **"первый.*последний"**

Результат: **"Первый может стать как последний и последний"**

Шаблон: **"первый.*?последний"**

Результат: **"Первый может стать как последний"**

Класи символів

Для поиска в регулярных выражениях можно задавать **классы символов**, заключенные в квадратные скобки. Во время поиска все символы в классе рассматриваются как один символ. Внутри класса можно задавать диапазон символов, помещая дефис между границами диапазона.

Если первым символом класса является знак вставки "^", то значение выражения инвертируется, такому классу соответствует **любой символ, не входящий в класс.**

Так как в классах символы "]", "^" и "-" имеют специальное значение, для их использования в классе существуют определенные правила:

- литерал "^" не должен быть первым символом класса;
- перед литералом "]" должен стоять символ обратной косой черты;
- для помещения в класс символа "-" достаточно либо поставить его на первую позицию, либо поместить перед ним символ обратной косой черты.

Примеры задания классов символов

"[абвг]" или **"[а-г]"**

Строка "**огонь**" удовлетворяет шаблону, поскольку в ней есть символ "г", а строка "**окно**" – не удовлетворяет, поскольку в ней нет ни одного из символов шаблона.

"Глава [0-9]+"

Строки "**Глава 5**" и "**Глава 18**" удовлетворяет шаблону, поскольку в них после строки "Глава" и пробела следуют цифры, а строка "**Глава десять**" – не удовлетворяет, так как в ней после слова "Глава" и пробела нет цифр.

"[А-Я][а-я]+"

Строка "**Иванов**" удовлетворяет шаблону, так как она начинается с заглавной буквы, за которой следуют строчные буквы, а строка "**ивановский**" – не удовлетворяет, поскольку она начинается со строчной буквы.

"[.?!]"

Строки "Как дела?", "Замечательно!" и "Хорошо." удовлетворяет шаблону, поскольку они содержат символы окончания предложения. Символы "." и "?" здесь используются как обычные символы, а не как метасимволы.

Спеціальні символи

Наиболее распространенные классы символов можно задать с помощью следующих специальных символов:

\d – соответствует любому цифровому символу (эквивалентно **[0-9]**);

\D – соответствует любому нецифровому символу (эквивалентно **[^0-9]**);

\w – соответствует любой латинской букве или цифре (эквивалентно **[A-Za-z0-9]**);

\W – соответствует любому небуквенному (латинскому) и нецифровому символу (эквивалентно **[^A-Za-z0-9]**);

\s – соответствует любому пробельному символу (эквивалентно **[\f\n\r\t\v]**);

\S – соответствует любому непробельному символу (эквивалентно **[^\f\n\r\t\v]**).

Специальные символы **\w** и **\W** нельзя использовать для букв алфавитов, отличных от латинских букв. В этом случае необходимо напрямую задавать диапазон символов, как это делается для классов символов.

Пример использования классов символов

Шаблон для номера мобильного телефона имеет следующий вид:

```
"\\d{3}-\\d{3}-\\d\\d-\\d\\d"
```

Этому шаблону соответствует телефонный номер

067-745-12-18

и не соответствует номер

055-867-1567

так как в нем нет тире перед предпоследней цифрой номера.

Анкери

С помощью анкеров можно указать, в каком месте строки должно быть найдено соответствие с шаблоном:

^ – соответствует позиции в начале строки;
$ – соответствует позиции в конце строки;
\b – соответствует границе между словом и пробельным символом;
\B – соответствует не границе слова.
Анкеры **\b** и **\B** действуют только для строк, состоящих из латинских букв.

Групування елементів

Операция группировки элементов в круглые скобки позволяет рассматривать данную группу элементов как один элемент.
Если в регулярных выражениях используются скобки, части искомой строки, соответствующие фрагментам в скобках, запоминаются в специальных переменных **$1** (первый фрагмент в скобках), **$2** (второй фрагмент в скобках), **$3** и т.д. Такая операция называется **захватом** (capture) **переменной**.

Пример использования анкеров

Шаблон **"^Глава \\d{1,2}\..*"**

Ищет в исходной строке следующие соответствия: строка "**Глава**" в начале строки, затем пробел, затем одна или две цифры, затем точка, затем любое содержимое до конца строки.

Пример использования группировки элементов

Необходимо найти в строке одно из слов:

белый, **красный**, **зеленый**, **желтый** или **черный**

Шаблон: **"белый|красный|зеленый|желтый|черный"**

либо: **"(бел|красн|зелен|желт|черн)ый"**

"(бел|желт|(крас|зеле|чер)н)ый".

Клас Pattern

Объект класса **Pattern** является откомпилированным представлением шаблона регулярного выражения и создается не с помощью ключевого слова **new**, а с помощью статических методов **compile()** класса **Pattern**.

Методи класу Pattern:

public static Pattern compile(String *шаблон)*

возвращает объект класса **Pattern** для заданного в параметре *шаблона*

public static Pattern compile(String *шаблон, int флажки)*

возвращает объект класса **Pattern** для заданного в параметре *шаблона с заданными флажками*

Флажки представлены **в** Java как статические поля типа **int** класса **Pattern (public static final int):**

CASE_INSENSITIVE – включает поиск соответствия без учета верхнего или нижнего регистра;

UNICODE_CASE – если этот флажок включен вместе с флажком **CASE_INSENSITIVE**, то верхний и нижний регистры букв в коде Unicode не учитываются при поиске соответствия;

UNIX_LINES –только символ "\n" учитывается как символ окончания строки, в которой выполняется поиск соответствия;

MULTILINE– если внутри строки, в которой выполняется поиск соответствия, есть символы "\n", то считается что строка состоит из нескольких строк;

LITERAL – все символы шаблона, включая метасимволы, рассматриваются как обычные символы;

DOTALL – если в шаблоне есть метасимвол ".", то ему будет соответствовать любой символ, включая символ "\n";

COMMENTS – в строке шаблона, допустимы пробелы и комментарии, начинающиеся с символа "#" до конца строки;

CANON_EQ – при поиске соответствия будет учитываться соответствие между кодом символа и сами символом, т.е. при включенном флажке латинская буква "a" будет соответствовать коду Unicode этой буквы "\u00E5" в шаблоне.

Если необходимо задать одновременно несколько флажков, то они должны быть разделены знаком операции ИЛИ – "|".

Пример

```
Pattern pattern1 =  Pattern.compile("abc");
```

Задание шаблона – строки **"abc".**

```
Pattern pattern2 =  Pattern.compile("string",
Pattern.CASE_INSENSITIVE);
```

Задание шаблона – строки **"string"** с поиском соответствия без учета регистра.

Флажки можно включать **непосредственно в шаблоне**, используя следующую синтаксическую форму:

(?строка-символов)

символы в ***строке-символов*** могут иметь одно из следующих значений

i – для флажка **CASE_INSENSITIVE**;

d – для флажка **UNIX_LINES**;

m – для флажка **MULTILINE**;

s – для флажка **DOTALL**;

u – для флажка **UNICODE_CASE**;

x – для флажка **COMMENTS**.

Пример

"(?ium)компьютер"

Методи класу Pattern

public static boolean matches(String *шаблон*, **CharSequence** *строка-поиска*)

проверяет соответствие ***шаблона строке-поиска*** и возвращает значение **true**, если строка поиска соответствует шаблону и **false** – в противном случае.

public String pattern()

возвращает строку шаблона для объекта класса **Pattern**;

public int flags()

возвращает числовое значения флажка для объекта класса **Pattern**; (если задано несколько флажков, возвращает сумму их числовых значений);

public static String quote(String *строка*)

возвращает строковый шаблон для заданной ***строки***;

Методи класу Pattern

public String[] split(CharSequence *строка-поиска*)

создает из ***строки-поиска*** массив, разделенный на элементы по шаблону, заданному в объекте класса **Pattern**;

public String[] split(CharSequence *строка-поиска*, int *предел*)

создает из ***строки-поиска*** массив, разделенный на элементы по шаблону, заданному в объекте класса **Pattern**, и с заданным в параметре ***предел*** количеством элементов. Если значение параметра больше или равно количеству элементов, либо меньше 0, выводятся все элементы, если меньше количества элементов – все оставшиеся соответствия выводятся в последнем элементе массива;

public String toString()

возвращает строковое представление откомпилированного шаблона.

Клас Matcher

Класс Matcher обеспечивает выполнение поиска или замены соответствия заданному объектом класса **Pattern** шаблону.

Объект класса **Matcher** создается с помощью метода

public Matcher matcher(CharSequence *строка-поиска)*

класса **Pattern** для *строки-поиска*.

Строковое представление объекта класса **Matcher** можно получить с помощью метода

public String toString()

Операції з регіонами

Поиск соответствия выполняется в подстроке исходной строки, называемой **регионом** (region). По умолчанию регионом является вся вводимая последовательность символов.

Установка границ региона

public Matcher region(int *начальный-индекс*, **int** *конечный-индекс*)

метод возвращает объект класса **Matcher** для подстроки, начинающейся с ***начального-индекса*** и заканчивающуюся индексом, на единицу меньшим, чем ***конечный-индекс***.

Текущие значения начального и конечного индексов региона

public int regionStart()
public int regionEnd().

Характер границ региона

public Matcher useAnchoringBounds(boolean *флажок*)
public Matcher useTransparentBounds(boolean *флажок*)

Методи пошуку відповідностей

public boolean matches()

выполняет для объекта класса **Matcher** поиск на соответствие всего региона, начиная с начала региона. Возвращает **true**, если соответствие найдено и **false** – в противном случае.

public boolean lookingAt()

выполняет для объекта класса **Matcher** поиск, начиная с начала региона на наличие шаблона в регионе, но необязательно соответствия всего региона шаблону. Возвращает **true**, если соответствие найдено и **false** – в противном случае.

public boolean find()

выполняет для объекта класса **Matcher** поиск, начиная с начала региона или, если предыдущий вызов метода был успешным, и объект класса **Matcher** не был сброшен, с первого символа после найденного предыдущего соответствия. Возвращает **true**, если соответствие найдено и **false** – в противном случае.

public boolean find(int *начальный-индекс*)

выполняется так же, как и предыдущей метод, но поиск начинается не с начала региона, а заданного ***начального-индекса***. Если необходимо найти все соответствия шаблону в строке, начиная с ***начального-индекса***, то этот метод можно использовать только для поиска первого соответствия. Все остальные соответствия определяются с помощью метода **find()** без параметров.

Методи заміни

Методы класса **Matcher** позволяют не только выполнить поиск в строке по заданному шаблону, но и заменить найденные соответствия заданными последовательностями символов – строками замены.

public String replaceFirst(String *строка-замены***)**

public String replaceAll(String *строка-замены* **)**

позволяют заменить только первое соответствие или все соответствия в строке поиска ***строкой-замены***. Оба метода возвращают измененную строку.

public StringBuffer appendTail(StringBuffer *новая-строка***)**

пересылает символы строки поиска в ***новую-строку***, начиная с конечной позиции и до конца строки поиска. Этот метод используется вместе с методом **appendReplacement()** для завершения процесса поиска и замены в строке.

public Matcher appendReplacement(StringBuffer *новая-строка*, **String** *строка-замены*)

формирует *новую-строку* по следующему алгоритму:

- пересылает символы строки поиска в ***новую-строку***, начиная с конечной позиции (append position) до символа на единицу меньшего, чем символ, определяемого методом **start()** объекта **Match**;
- к новой строке добавляется ***строка-замены***;
- конечная позиция в новой строке становится равной позицией, определяемой методом **end()** объекта **Match**.

В начале просмотра и замены значение конечной позиции равно 0.

Методы **appendReplacement()** и **appendTail()** выполняют те же действия, что и методы **replaceFirst()** и **replaceAll()**, однако они позволяют управлять как количеством замен, так самими заменами в строке.

Клас PatternSyntaxException

Класс **PatternSyntaxException** бросает исключение, если регулярное выражение (шаблон) содержит синтаксическую ошибку.

В классе определены следующие методы:

public String getPattern() возвращает шаблон, содержащий ошибку

public String getDescription() возвращает описание ошибки;

public int getIndex() возвращает позицию символа ошибки в шаблоне;

public String getMessage()

возвращает сообщение об ошибке, содержащее все перечисленные выше компоненты: описание ошибки и ее индекс, шаблон, содержащий ошибку и визуальную индикацию индекса ошибки внутри шаблона.

Методи класу String
для роботи з регулярними виразами

public boolean matches(String *шаблон*)

если объект класса **String** соответствует ***шаблону***, возвращает значение **true**, в противном случае возвращает **false** (действует аналогично методу **matches()** класса **Pattern**);

public String[] split(String *шаблон*)

создает для объекта класса **String** массив строк, разделенный на элементы по заданному ***шаблону*** (действует аналогично соответствующему методу **split()** класса **Pattern**);

public String[] split(String *шаблон*, int *предел*)

– создает для объекта класса **String** массив строк, разделенный на элементы по заданному ***шаблону***, и с заданным в параметре ***предел*** количеством элементов (если значение параметра больше или равно количеству элементов, либо меньше 0, выводятся все элементы, если меньше количества элементов – все оставшиеся соответствия выводятся в последнем элементе массива) (действует аналогично соответствующему методу **split()** класса **Pattern**);

public String replaceFirst(String *шаблон*, String *строка-замены*)

заменяет в объекте **String** первое соответствие ***шаблону*** на ***строку-замены*** и возвращает измененную строку (действует аналогично методу **replaceFirst ()** класса **Match**);

public String replaceAll(String *шаблон*, String *строка-замены*)

заменяет в объекте **String** все соответствия ***шаблону*** на ***строку-замены*** и возвращает измененную строку (действует аналогично методу **replaceAll()** класса **Match**).