**DYNAMIC FIND**

A feature that helps user to find characters or words in the text typed in the editor and highlights them. Allows user to search using **regular expressions.**

**Regular expression**

In theoretical computer scienceand formal language theory, a regular expression is a sequence of characters that forms a search pattern. Mainly for use in pattern matching with strings or string matching, i.e. “find and replace”-like operations.

Each character in the regular expression is either understood to be a metacharacter with its special meaning, or a regular character with its literal meaning. Together, they can be used to identify textual material of a given pattern, or process a number of instances of it that can vary from a precise equality to a very general similarity of the pattern. The pattern sequence itself is an expression that is a statement in a language designed specifically to represent prescribed targets in the most concise and flexible way to direct the automation of text processing of general text files, specific textual forms, or of random input strings.

A **metacharacter** is a character that has a special meaning (instead of a literal meaning) to a computer program, such as a shell interpreter or a regular expression engine. In regular expressions, there are 12 metacharacters that must always be preceded by a backslash, \, to be used inside of the expression: The opening square bracket [, the closing square bracket], the backslash \, the caret ^, the dollar sign $, the period or dot, the vertical bar or pipe symbol |, the question mark?, the asterisk or star \*, the plus sign +, the opening round bracket (and the closing round bracket).

If you want to use any of these characters as a literal in a regex, you need to escape them with a backslash. If you want to match 1+1=2, the correct regex is 1\+1=2. Otherwise, the plus sign will have a special meaning.

|  |  |
| --- | --- |
| METACHARACTER | Description |
| . | Matches any character |
| [ ] | Matches a single character that is contained within the brackets |
| [^ ] | Matches a single character that is not contained within the brackets. |
| ^ | Matches the starting position within the string |
| $ | Matches the ending position of the string or the position just before a string-ending newline. |
| \* | Matches the preceding element zero or more times |
| ? | Matches the preceding element zero or one time. |
| + | Matches the preceding element one or more times |
| | | The choice (also known as alternation or set union) operator matches either the expression before or the expression after the operator |
| {m,n} | Matches the preceding element at least *m* and not more than *n* times |
| ( ) | Defines a marked sub expression |

**Examples**

* .at matches any three-character string ending with "at", including "hat", "cat", and "bat".
* [hc]at matches "hat" and "cat".
* [^b]at matches all strings matched by .at except "bat".
* [^hc]at matches all strings matched by .at other than "hat" and "cat".
* ^[hc]at matches "hat" and "cat", but only at the beginning of the string or line.
* [hc]at$ matches "hat" and "cat", but only at the end of the string or line.
* \[.\] matches any single character surrounded by "[" and "]" since the brackets are escaped, for example: "[a]" and "[b]".
* s.\* matches any number of characters preceded by s, for example: "saw" and "seed".

**Implementation**

Using various in - built packages, classes and their utilities (functions) of java along with java swing components and event listeners.

**Packages used**

* javax.swing : For swing components.
* javax.swing.text : For swing text features.
* java.awt : Abstract window toolkit package, for creating UI.
* java.awt.event : For events and listeners.
* java.util.regex : Regex package

Apart from these we make use of **Highlighter** interface, **Matcher** and **Pattern** classes.

* javax.swing.text.Highlighter
* java.util.regex.Matcher
* java.util.regex.Pattern

**Finding position of the character or word**

To find a specified character or word in the text, javax.util.regex package is used, which provides necessary classes and functions.

1. **Pattern:** A compiled representation of regular expression.

**public Pattern compile( String ):** Used to create instance of the pattern.

1. **Matcher:** An engine that performs match operations on a character sequence by interpreting a pattern.

**public Matcher matcher( String ):** Used to match pattern again text.

**public Boolean find():** returns true if match is found.

**public String group():** returns input sequence matched.

**public int start():** returns index of the match.

**Code Snippet**

**/\***

Pattern p=Pattern.compile(str); // **str: pattern to be matched.**

String typo=t.getText(); **// typo: text typed in the editor.**

Matcher m=p.matcher(typo);

**\*/**

**To highlight the characters or words**

To highlight the words found by the matcher, Highlighter interface utilities are used.

**public interface Highlighter**

**public Object addHighlight(int p0,int p1,Highlighter.HighlightPainter p):** Used to highlight words.

**public void removeAllHighlight():** Used to remove all highlights.

**public Highlighter getHighlighter():** Used to get Highlighter.

**Code Snippet**

**/\***

while(m.find()){

try { h.addHighlight(m.start(),m.start()+m.group().length(),DefaultHighlighter.DefaultPainter);

} catch (BadLocationException e1) {

e1.printStackTrace();

}

\*/

**addHighlight()** function throws an exception(BadLocationException).To catch the exception, the function is surrounded within try and catch block.

* Using start() function, the position of the word in the text area, which is then passed as a parameter to the function addHighlight() as shown in the snippet above.
* addHighlight(), highlights the word using default option(BLUE in colour).

**REPLACE**

An option that allows user to replace a word or a character (multiple occurrences) with another.

**Implementation**

Implemented using default packages of java and its utilities in a **Brute force approach.**

**Brute Force**

In computer science, **brute-force search** or **exhaustive search**, also known as **generate and test**, is a very general problem-solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement, some of the examples are;

1. Linear search
2. Selection sort
3. Bubble sort

**Packages used**

* **java.util.regex.Matcher:** Provides matching operations.
* **java.util.regex.Pattern :** Provide functions for creating compiled versions of pattern.

**To find position of a word or character**

To find a specified character or word in the text, javax.util.regex package is used, which provides necessary classes and functions.

1. **Pattern:** A compiled representation of regular expression.

**public Pattern compile( String ):** Used to create instance of the pattern.

1. **Matcher:** An engine that performs match operations on a character sequence by interpreting a pattern.

**public Matcher matcher( String ):** Used to match pattern again text.

**public Boolean find():** returns true if match is found.

**public String group():** returns input sequence matched.

**public int start():** returns index of the match.

**Code Snippet**

**/\***

str=(String)t1.getText();

h.removeAllHighlights();

Pattern p=Pattern.compile(str);

String typo=t.getText();

Matcher m=p.matcher(typo);

while(m.find()){

ar[i]=m.start(); **/\* array ar[] holds position of the word or character \*/**

System.out.println(ar[i]);

j=i+1; **// j holds length of the array**

i++;

}

\*/

**To replace with the specified word or character**

To replace, we use brute force approach where in at each position of the word or character to be replaced in the text area, we collect the text before the position and after the position and store it. At the position, we replace the word or character with another (replacement).Later, we concatenate the entire text. The process is repeated at each position of the word or character to be replaced.

* To collect the text before and after the position we make use of **substring( int index )** function of **String,** with appropriate arguments.
* To concatenate and set it to text area we use **setText( String )** function of TextArea.

**Code Snippet**

**/\***

for(i=0;i<j;i++)

{

String before= t.getText().substring(0,ar[i]+i\*(t2.getText().length()-t1.getText().length()));

String after= t.getText().substring(ar[i]+i\*(t2.getText().length()-

t1.getText().length())+t1.getText().length());

t.setText(before+t2.getText()+after);

}

**\*/**

**Examples**

**substring(0,4):** retrieves substring from position 0 to position 3.

**substring(4):** retrieves substring from position 4 to end of the string.