Pattern Matching with Regular Expressions

What are regular expressions?

- A regular expression is a sequence of characters.
- The abbreviation for regular expression is *regex*.
- Java Regular Expressions Package, ***java.util.regex.****
- A *regular expression* defines a search pattern for strings.
- The search pattern can be anything, such as,
    - a simple character, or
    - a fixed string or
    - a complex expression containing special characters describing the pattern.
- The pattern defined by the regex may match one or several times or not at all for a given string.
- Regular expressions can be used to search, edit and manipulate string ("find and replace"-like operations.), URL matching, etc.
- The pattern defined by the regex is applied on the text from left to right.
    - For example, the regex aba will match ***aba****bab****aba*** only two times (aba_aba__).

Regular expression metacharacter syntax

Subexpression	**Matches**	**Notes**
General		
\^	Start of line/string	
$	End of line/string	
\b	Word boundary	
\B	Not a word boundary	
\A	Beginning of entire string	
\z	End of entire string	
\Z	End of entire string (except allowable final line terminator)	
.	Any one character (except line terminator)	
[…]	"Character class"; any one character from those listed	
[\^…]	Any one character not from those listed	
Alternation and Grouping		
(…)	Grouping (capture groups)	
|	Alternation	
(?:_re_)	Noncapturing parenthesis	
\G	End of the previous match	

Subexpression	**Matches**	**Notes**
\ *n*	Back-reference to capture group number "*n*"	

Normal (greedy) quantifiers

{ *m*,*n* }	Quantifier for “from *m* to *n* repetitions”	
{ *m* ,}	Quantifier for "*m* or more repetitions”	
{ *m* }	Quantifier for “exactly *m* repetitions”	
{,*n* }	Quantifier for 0 up to *n* repetitions	
*	Quantifier for 0 or more repetitions	Short for {0,}
+	Quantifier for 1 or more repetitions	Short for {1,}
?	Quantifier for 0 or 1 repetitions (i.e., present exactly once, or not at all)	Short for {0,1}

Reluctant (non-greedy) quantifiers

{ *m*,*n* }?	Reluctant quantifier for “from *m* to *n* repetitions”	
{ *m* ,}?	Reluctant quantifier for "*m* or more repetitions”	
{,*n* }?	Reluctant quantifier for 0 up to *n* repetitions	
*?	Reluctant quantifier: 0 or more	
+?	Reluctant quantifier: 1 or more	
??	Reluctant quantifier: 0 or 1 times	

Subexpression	**Matches**	**Notes**
Possessive (very greedy) quantifiers		
{ *m*,*n* }+	Possessive quantifier for “from *m* to *n* repetitions”	
{ *m* ,}+	Possessive quantifier for "*m* or more repetitions”	
{,*n* }+	Possessive quantifier for 0 up to *n* repetitions	
*+	Possessive quantifier: 0 or more	
++	Possessive quantifier: 1 or more	
?+	Possessive quantifier: 0 or 1 times	
Escapes and shorthands		
\	Escape (quote) character: turns most metacharacters off; turns subsequent alphabetic into metacharacters	
\Q	Escape (quote) all characters up to \E	
\E	Ends quoting begun with \Q	
\t	Tab character	
\r	Return (carriage return) character	
\n	Newline character	
\f	Form feed	

Subexpression	**Matches**	**Notes**
\w	Character in a word	Use \w+ for a word;
\W	A nonword character	
\d	Numeric digit	Use \d+ for an integer;
\D	A nondigit character	
\s	Whitespace	Space, tab, etc., as determined by java.lang.Character.isWhitespace()
\S	A nonwhitespace character	

Unicode blocks (representative samples)

\p{InGreek}	A character in the Greek block	(Simple block)
\P{InGreek}	Any character not in the Greek block	
\p{Lu}	An uppercase letter (Simple category)	
\p{Sc}	A currency symbol	

POSIX-style character classes (defined only for US-ASCII)

\p{Alnum}	Alphanumeric characters	[A-Za-z0-9]
\p{Alpha}	Alphabetic characters	[A-Za-z]
\p{ASCII}	Any ASCII character	[\x00-\x7F]
\p{Blank}	Space and tab characters	
\p{Space}	Space characters	[\t\n\x0B\f\r]
\p{Cntrl}	Control characters	[\x00-\x1F\x7F]

Subexpression	Matches	Notes
\p{Digit}	Numeric digit characters	[0-9]
\p{Graph}	Printable and visible characters (not spaces or control characters)	
\p{Print}	Printable characters	
\p{Punct}	Punctuation characters	One of !"#$%&'()*+,-./:;<=>?@[]\^_`{|}\~
\p{Lower}	Lowercase characters	[a-z]
\p{Upper}	Uppercase characters	[A-Z]
\p{XDigit}	Hexadecimal digit characters	[0-9a-fA-F]

How to write regular expression?

- **Repeaters : * , + and { } :**
  - These symbols act as repeaters and tell the computer that the preceding character is to be used for more than just one time.
  -
  - **The asterisk symbol (*):**
    It tells the computer to match the preceding character (or set of characters) for 0 or more times (upto infinite).
    - **Example :** The regular expression ab*c will give ac, abc, abbc, abbbc….ans so on
  -
  - **The Plus symbol (+):**

    It tells the computer to repeat the preceding character (or set of characters) for atleast one or more times(upto infinite).

    - **Example :** The regular expression ab+c will give abc, abbc, abbbc, … and so on.

    - **The curly braces {…}:**

      It tells the computer to repeat the preceding character (or set of characters) for as many times as the value inside this bracket.

        - **Example :** {2} means that the preceding character is to be repeated 2 times,
        - {min,} means the preceding character is matches min or more times.

        - {min,max} means that the preceding character is repeated at least min & at most max times.

- **Wildcard – (.)**
  The dot symbol can take place of any other symbol, that is why it

  is called the wildcard character.

    - **Example :** The Regular expression .* will tell the computer that any character can be used any number of times.
- **Optional character – (?)**

  This symbol tells the computer that the preceding character may or may not be present in the string to be matched.

    - **Example :** We may write the format for document file as – "doc**x**?"
        - The '?' tells the computer that x may or may not be present in the name of file format.

- **The caret (^) symbol:**

  Setting position for match : tells the computer that the match must start at the beginning of the string or line.

        - **Example :** ^\d{3} will match with patterns like "901" in "901-333-".

- **The dollar ($) symbol**
  It tells the computer that the match must occur at the end of the string or before \n at the end of the line or string.
  - **Example :** -\d{3}$ will match with patterns like "-333" in "-901-333".
- **[set_of_characters]** – Matches any single character in set_of_characters. By default, the match is case-sensitive.
  - **Example :** [abc] will match characters a, b or c in any string.

- **[^set_of_characters] –***Negation:* Matches any single character that is not in set_of_characters. By default, the match is case sensitive.
  - **Example :** [^abc] will match any character except a,b,c.

    .
- **[first-last]** – *Character range:* Matches any single character in the range from first to last.
  - **Example :** [a-zA-z] will match any character from a to z or A to Z.

- **The Escape Symbol : **
  If you want to match for the actual '+', '.' etc characters, add a backslash(\ ) before that character. This will tell the computer to treat the following character as a search character and consider it for matching pattern.
  - **Example :** \d+[\+-x*]\d+ will match patterns like "2+2" and "3*9" in "(2+2) * 3*9".

- **Grouping Characters ( )**
  A set of different symbols of a regular expression can be grouped together to act as a single unit and behave as a block, for this, you need to wrap the regular expression in the parenthesis( ).
  - **Example :** ([A-Z]\w+) contains two different elements of the regular expression combined together. This expression will match any pattern containing uppercase letter followed by any character.

- **Vertical Bar (|) :**
  Matches any one element separated by the vertical bar (|) character.

- **Example :** th(e|is|at) will match words - the, this and that.

Example: There are three ways to write the regex example in Java.

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

public class Regex {

        public static void main(String[] args) {

                //1st way
                Pattern p = Pattern.compile(".s");  // . represents single
                                                        character
                Matcher m = p.matcher("as");
                boolean b = m.matches();

                //2nd way
                boolean b2=Pattern.compile(".s").matcher("as").matches();

                //3rd way
                boolean b3 = Pattern.matches(".s", "as");

                System.out.println(b+""+b2+""+b3);
        }
}
```

Output:

```
true    true    true
```

Example: Regular Expression . (dot), which represents a single character.

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

public class Regex {

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

```
		System.out.println(Pattern.matches(".s", "as"));
								//true (2nd char is s)

		System.out.println(Pattern.matches(".s", "mk"));
							//false (2nd char is not s)

		System.out.println(Pattern.matches(".s", "mst"));
						//false (has more than 2 char)

		System.out.println(Pattern.matches(".s", "amms"));
							//false (has more than 2 char)

		System.out.println(Pattern.matches("..s", "mas"));
							//true (3rd char is s)
	}
}
```

Output:

```
true
false
false
false
true
```

Example: Regular Expression Character classes

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

public class Regex {

	public static void main(String[] args) {

		System.out.println(Pattern.matches("[amn]", "abcd"));
							//false (not a or m or n)

		System.out.println(Pattern.matches("[amn]", "a"));
							//true (among a or m or n)
```

```
        System.out.println(Pattern.matches("[amn]", "ammmna"));
                                        //false (m and a comes more than once)

        System.out.println(Pattern.matches("[^n].*", "abc"));//true

        System.out.println(Pattern.matches("[^n]", "abc mnc"));
                                                        //false
    }
}
```

Output:

```
false
true
false
true
false
```

Example: Regular Expression Character classes and Quantifiers

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

public class Regex {

    public static void main(String[] args) {

        System.out.println("? quantifier ....");

        System.out.println(Pattern.matches("[amn]?", "a"));
                                            //true (a or m or n comes one time)

        System.out.println(Pattern.matches("[amn]?", "aaa"));
                                            //false (a comes more than one time)

        System.out.println(Pattern.matches("[amn]?", "aammmnn"));
                                        //false (a m and n comes more than one time)

        System.out.println(Pattern.matches("[amn]?", "aazzta"));
                                        //false (a comes more than one time)
```

```
                System.out.println(Pattern.matches("[amn]?", "am"));
                                        //false (a or m or n must come one time)

                System.out.println("+ quantifier ....");

                System.out.println(Pattern.matches("[amn]+", "a"));
                                                //true (a or m or n once or more times)

                System.out.println(Pattern.matches("[amn]+", "aaa"));
                                                //true (a comes more than one time)

                System.out.println(Pattern.matches("[amn]+", "aammmnn"));
                                                //true (a or m or n comes more than once)
                System.out.println(Pattern.matches("[amn]+", "aazzta"));
                                                //false (z and t are not matching pattern)

                System.out.println("* quantifier ....");

                System.out.println(Pattern.matches("[amn]*", "ammmnaa"));
                                        //true (a or m or n may come zero or more times)
        }
}
```

Output:

```
? quantifier ....
true
false
false
false
false
+ quantifier ....
true
true
true
false
* quantifier ....
true
```

Example: Regular Expression Metacharacters, \d and \D

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

public class Regex {

	public static void main(String[] args) {

		System.out.println("metacharacters d....");    //d means digit

		System.out.println(Pattern.matches("\\d", "abc"));
		                                                  //false (non-digit)

		System.out.println(Pattern.matches("\\d", "9"));
		                                    //true (digit and comes once)

		System.out.println(Pattern.matches("\\d", "4443"));
		                         //false (digit but comes more than once)


		System.out.println(Pattern.matches("\\d", "323abc"));
		                                                  //false (digit and char)

		System.out.println(Pattern.matches("\\d*", "3"));
		                               //true (digit and comes more than once)

		System.out.println("metacharacters d with quantifier....");

		System.out.println("metacharacters D....");
		                                                  //D means non-digit

		System.out.println(Pattern.matches("\\D", "abc"));
		                         //false (non-digit but comes more than once)

		System.out.println(Pattern.matches("\\D", "1"));
		                                                          //false (digit)

		System.out.println(Pattern.matches("\\D", "4443"));
		                                                          //false (digit)
```

```
                System.out.println(Pattern.matches("\\D", "323abc"));
                                                                //false (digit and char)

                System.out.println(Pattern.matches("\\D", "m"));
                                                    //true (non-digit and comes once)

                System.out.println("metacharacters D with quantifier....");

                System.out.println(Pattern.matches("\\D*", "mak"));
                                //true (non-digit and may come 0 or more times)
        }
}
```

Output:

```
metacharacters d....
false
true
false
false
metacharacters d with quantifier....
true
metacharacters D....
false
false
false
false
true
metacharacters D with quantifier....
true
```

Example: Create a regular expression that accepts alphanumeric characters only. It ’s length must be six characters long only.

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

public class Regex1 {

        public static void main(String[] args) {
                System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun2"));
                                                                                    //true

                System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "kkvarun32"));
                                                                  //false (more than 6 char)

                System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "JA2Uk2"));
                                                                                    //true

                System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun$2"));
                                                                  //false ($ is not matched)
        }
}
```

Output:

```
true
false
true
false
```

Example: Create a regular expression that accepts 10 digit numeric characters starting with 7, 8 or 9 only.

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

public class Regex1 {

        public static void main(String[] args) {

                System.out.println("by character classes and quantifiers ...");

                System.out.println(Pattern.matches("[789]{1}[0-9]{9}",
                                                        "9953038949")); //true

                System.out.println(Pattern.matches("[789][0-9]{9}",
                                                        "9953038949")); //true
```

```
                    System.out.println(Pattern.matches("[789][0-9]{9}",
                                   "99530389490"));  //false (11 characters)

                     System.out.println(Pattern.matches("[789][0-9]{9}",
                                        "6953038949"));//false (starts from 6)

                    System.out.println(Pattern.matches("[789][0-9]{9}",
                                                       "8853038949"));//true

                    System.out.println("by metacharacters ...");

                    System.out.println(Pattern.matches("[789]{1}\\d{9}",
                                                       "8853038949"));//true

                    System.out.println(Pattern.matches("[789]{1}\\d{9}",
                                        "3853038949")); //false (starts from 3)
            }
}
```

Output:

```
by character classes and quantifiers ...
true
true
false
false
true
by metacharacters ...
true
false
```

Example: Subexpression: ^ Matches: Start of line/string

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

public class Regex1 {

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

```
                    String Regex = "^dog.*";
                    String Input = "dog xyz";

                    boolean match = Pattern.matches(Regex, Input);

                    System.out.print("Output: ");

                    if (match)
                        System.out.print(" found");
                    else
                        System.out.print(" not found");
        }
}
```

Output: found

Example: Subexpression: $ Matches: End of line/string

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

public class Regex1 {

        public static void main(String[] args) {

                    String Regex = ".*dog$";
                    String Input = "abc dog";

                    boolean match = Pattern.matches(Regex, Input);

                    System.out.print("Output: ");

                    if (match)
                        System.out.print(" found");
                    else
                        System.out.print(" not found");
        }
}
```

Output: found

Example: Subexpression: \b Matches: Word boundary

Xyz abc pqr

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

public class Regex1 {

	public static void main(String[] args) {

				String Regex = ".*\\bdog\\b.*";
				String Input = "abc dog xyz";

				//String Input = "abcdogggxyz"; //Output:  not found


				boolean match = Pattern.matches(Regex, Input);

				System.out.print("Output: ");
				if (match)
					System.out.print(" found");
				else
					System.out.print(" not found");

	}

}
```

Output: found

Example: Subexpression: \B Matches: Not a word boundary

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

public class Regex1 {

	public static void main(String[] args) {

				String Regex = ".*\\bdog\\B.*";
						// \b word boundary \B not word boundary
				String Input = "abc dogpqr xyz";
```

```
                    //String Regex = ".*\\Bgpq\\B.*";
                                                  // \B not word boundary
                    //String Input = "abcdogpqrxyz";

                    booleanmatch = Pattern.matches(Regex, Input);

                    System.out.print("Output: ");
                    if (match)
                              System.out.print(" found");
                    else
                              System.out.print(" not found");
          }
}
```

Output: found

Example: Subexpression: \A Matches: Beginning of entire string

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

public class Regex1 {

          public static void main(String[] args) {

                    String Regex = "\\Aabc.*";
                    String Input = "abc dogpqr xyz";
                    //String Regex = ".*\\Axyz";      // Output: not found
                    //String Input = "abcdogpqrxyz";
                    boolean match = Pattern.matches(Regex, Input);
                    System.out.print("Output: ");
                    if (match)
                              System.out.print(" found");
                    else
                              System.out.print(" not found");
          }
}
```

Output: found

Example: Subexpression: \z Matches: End of entire string

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

public class Regex1 {

	public static void main(String[] args) {

			String Regex = ".*xyz\\z";
			String Input = "abc dogpqr xyz";

			boolean match = Pattern.matches(Regex, Input);

			System.out.print("Output: ");
			if (match)
				System.out.print(" found");
			else
				System.out.print(" not found");
	}
}
```

Output: found

Example: Subexpression: [...] Matches: "Character class"; any one character from those listed string

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

public class Regex1 {

	public static void main(String[] args) {

			String Regex = ".*a[bc]d.*";
			String Input1= "pqr abd xyz";
			String Input2 = "pqr acd xyz";

			//String Input = "abcabcdxyz";Output:  not found

			boolean match1 = Pattern.matches(Regex, Input1);
			boolean match2 = Pattern.matches(Regex, Input2);
```

```
                System.out.print("Output: ");
                if (match1)
                        System.out.println(" found");
                else
                        System.out.println(" not found");

                if (match2)
                        System.out.println(" found");
                else
                        System.out.println(" not found");
        }
}
```

Output: found

found

Example: Subexpression: [\^...] Matches: Any one character not from those listed

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

public class Regex1 {

        public static void main(String[] args) {

                String Regex = ".*a[b^c]d.*";
                String Input = "abc aed xyz";
                //String Regex = ".*ab[^c]d.*"; Output:  not found
                boolean match = Pattern.matches(Regex, Input);
                System.out.print("Output: ");
                if (match)
                        System.out.print(" found");
                else
                        System.out.print(" not found");

        }

}
```

Output: found

Using regexes in Java: Test for a Pattern

- The java.util.regex package consists of two classes, Pattern and Matcher , which provide the public API shown as follows:

Example 4-1. Regex public API

```
/** The main public API of the java.util.regex package.
* Prepared by javap and Ian Darwin.
*/
package java.util.regex;
public final class Pattern {
        // Flags values ('or' together)
        public static final int
                UNIX_LINES, CASE_INSENSITIVE, COMMENTS, MULTILINE,
                DOTALL, UNICODE_CASE, CANON_EQ;
        // No public constructors; use these Factory methods
        public static Pattern compile(String patt);
        public static Pattern compile(String patt, int flags);
        // Method to get a Matcher for this Pattern
        public Matcher matcher(CharSequence input);
        // Information methods
        public String pattern();
        public int flags();
        // Convenience methods
        public static boolean matches(String pattern, CharSequence input);
        public String[] split(CharSequence input);
        public String[] split(CharSequence input, int max);
}


public final class Matcher {
        // Action: find or match methods
        public boolean matches();
        public boolean find();
        public boolean find(int start);
        public boolean lookingAt();
        // "Information about the previous match" methods
        public int start();
        public int start(int whichGroup);
        public int end();
        public int end(int whichGroup);
        public int groupCount();
        public String group();
        public String group(int whichGroup);
        // Reset methods
        public Matcher reset();
        public Matcher reset(CharSequence newInput);
        // Replacement methods
        public Matcher appendReplacement(StringBuffer where, String newText);
        public StringBuffer appendTail(StringBuffer where);
        public String replaceAll(String newText);
        public String replaceFirst(String newText);
        // information methods
        public Pattern pattern();
}


/* String, showing only the RE-related methods */
public final class String {
        public boolean matches(String regex);
        public String replaceFirst(String regex, String newStr);
        public String replaceAll(String regex, String newStr);
        public String[] split(String regex);
        public String[] split(String regex, int max);
}
```

Matching regexes using Pattern and Matcher(s)

- If the regex is going to be used more than once or twice in a program, it is more efficient to construct and use a Pattern and its Matcher(s).
- *The normal steps for regex*

  1. Create a Pattern by calling the static method Pattern.compile().

  2. Request a Matcher from the pattern by calling pattern.matcher(CharSequence) for each String (or other CharSequence) you wish to look through.

3. Call (once or more) one of the finder methods (discussed later in this section) in the resulting Matcher.

The Matcher methods are:

matches()

Used to compare the entire string against the pattern; this is the same as the routine in java.lang.String. Because it matches the entire String, need to put .* before and after the pattern.

lookingAt()

Used to match the pattern only at the beginning of the string.

find()

Used to match the pattern in the string (not necessarily at the first character of the string), starting at the beginning of the string or, if the method was previously called and succeeded, at the first character not matched by the previous match.

Example: matches()

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

public class RESimple {

        public static void main(String[] args) {

                String pattern = ".*abd.*";

                String input ="pqr bad pxy";

                Pattern p = Pattern.compile(pattern);

                Matcher m=p.matcher(input);

                if(m.matches()) {

                        System.out.println("Patern "+ pattern + "found in string " +

                                                                        input);
```

```
            }
            Else {
                    System.out.println("Patern "+ pattern +  "not found in string "
                                                                        + input);
            }
        }
}
```

Output: Patern .*abd.* found in string pqr abd pxy

Example: Another example on matches()

```
import java.util.regex.*;
public class RESimple {
        public static void main(String[] args) {
            String pattern = ".*Q[^u]\\d+\\..*";
            String line = "Order QT300. Now!";
            if (line.matches(pattern)) {
                System.out.println(line + " matches \"" + pattern + "\"");
            }
          else {
                System.out.println("NO MATCH");
            }
        }
}
```

Output: Order QT300. Now! Matches ".*Q[^u]\d+\..*"

Example: lookingAt()

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

public class RESimple {

        public static void main(String[] args) {

                String pattern = "pqr";
                String input ="pqr abd pxy";

                Pattern p = Pattern.compile(pattern);

                Matcher m=p.matcher(input);

                if(m.lookingAt()){

                                System.out.println("Patern "+pattern+

                                                                " found in string "+input);

                }

                else {

                                System.out.println("Patern "+pattern+

                                                        " not found in string "+input);

                }

        }

}
```

Output: Patern pqr found in string pqr abd pxy

Example: Another example on lookingAt()

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

public class RESimple {

        public static void main(String[] args) {

                String pattern = "Q[^u]\\d+\\."; "Q[A-Za-z\\d]+[\\.,]"

                String[] input = { "QA777. is the next flight. It is on time.",

                                                        "Quack, Quack, Quack!"};

                Pattern p = Pattern.compile(pattern);

                for (String in : input) {

                        boolean found =p.matcher(in).lookingAt();

                        System.out.println("'" + pattern + "'" +

                             (found ? " matches '" : " doesn't match'") + in + "'");

                }

        }

}
```

Output: '^Q[^u]\d+\.' matches 'QA777. is the next flight. It is on time.'
'^Q[^u]\d+\.' doesn't match 'Quack, Quack, Quack!'

Example: find()

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

public class RESimple {

        public static void main(String[] args) {

                String pattern = "abd";
                String input ="pqr abd pxy";

                Pattern p = Pattern.compile(pattern);

                Matcher m=p.matcher(input);

                if(m.find()) {

                        System.out.println("Patern " + pattern +

                                                        "found in string "+input);

                }

                else {

                System.out.println("Patern " + pattern +

                                                    "not found in string "+input);

                }

        }

}
```

Output: Patern abd found in string pqr abd pxy

Example: Another example of find()

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

public class RESimple {

        public static void main(String[] args) {

                String patt = ".*Q[^u]\\d+\\..*";
                String line = "Order QT300. Now!";

                Matcher m = Pattern.compile(patt).matcher(line);

                if (m.find( )) {

                        System.out.println(line + " matches " + patt);
                }
                else {
                        System.out.println(line +"not matches " + patt);
                }
        }
}
```

Output: Order QT300. Now! matches .*Q[^u]\d+\..*

Finding the Matching Text

- If you need to know exactly what characters were matched.
- **start() and end()**
    - Returns the character position in the string of the starting and ending characters that matched.
    - The start() method returns the start index of first character of the subsequence(i.e., regex), matched into the string.
    - The end() method returns the index of the last character matched into the string, plus one.

Example:

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

public class Regex1 {

public static void main(String[] args) {

                String pattern = "abd";
                String input ="pqr abd pxy abd";
                Pattern p = Pattern.compile(pattern);
                Matcher m=p.matcher(input);
                int count=0;
                while(m.find()) {
                        count++;
                        System.out.println("The " + count + " " + pattern +"  Pattern
                                found from " + m.start() +         " to " + (m.end()-1));
                }
        }
}
```

Output:

```
The 1 abd  Pattern found from 4 to 6
The 2 abd  Pattern found from 12 to 14
```

groupCount()

- The java.time.Matcher.groupCount() method returns the number of capturing groups in this matcher's pattern.
- Returns the number of parenthesized capture groups, if any groups were used.
- Returns 0 if no groups were used.

Example:

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

public class GroupCount {

        public static void main(String[] args) {

           String regex = "(.*)(\\d+)(.*)";
           String input = "This is a sample Text, 1234, with numbers in between.";

           //Creating a pattern object
           Pattern pattern = Pattern.compile(regex);
```

```
                //Matching the compiled pattern in the String
                Matcher matcher = pattern.matcher(input);

                System.out.println("Number of groups capturing: "
                                                        + matcher.groupCount());

            }
}
```

Output:

```
Number of groups capturing: 3
```

Example: Another Example of groupCount()

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

public class Regex1 {

public static void main(String[] args) {

                    //String pattern = ".*\\d{6}"; //Output: Total group = 0
                    //String pattern = "(.*)(\\d{6})"; //Output: Total group = 2
                    String pattern = "(.*)(\\d{6})(.*)"; //Output: Total group = 3

                    String input ="abdpxy 100000";

                    Pattern p = Pattern.compile(pattern);
                    Matcher m=p.matcher(input);

                    System.out.println("Total group = "+m.groupCount());
          }
}
```

Output:

```
Total group = 3
```

group(int i)

- The group(int i) method retrieve the characters that matched a given parenthesis group.
- Returns the characters matched by group i of the current match, if i is greater than or equal to zero and less than or equal to the return value of groupCount() .
- Group 0 is the entire match, so group(0) (or just group()) returns the entire portion of the input that matched.
- If you haven't used any explicit parens, you can just treat whatever matched as "level zero."

Example:

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

public class GroupCount {

        public static void main(String[] args) {

                String regex = "(.*)(\\d+)(.*)";
                String input = "This is a sample Text, 1234, with numbers in between.";

               //Creating a pattern object
               Pattern pattern = Pattern.compile(regex);

               //Matching the compiled pattern in the String
                Matcher matcher = pattern.matcher(input);

                 if(matcher.find()) {
                         System.out.println("First group match: "+matcher.group(1));
                        System.out.println("Second group match: "+matcher.group(2));
                        System.out.println("Third group match: "+matcher.group(3));
                        System.out.println("Number of groups capturing: "
                                                                + matcher.groupCount());
        }
    }
}
```

Output:

```
First group match: This is a sample Text, 123
Second group match: 4
Third group match: , with numbers in between.
Number of groups capturing: 3
```

Example:

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

public class Regex1 {

    public static void main(String[] args) {

        String pattern = "(.*)(\\d{6})(.*)"; //Output: Total group = 3
        String input ="abdpxy 100000";

        Pattern p = Pattern.compile(pattern);
        Matcher m=p.matcher(input);

        int k = m.groupCount();
        System.out.println("Total group count= "+ k);

        while(m.find()) {
            for(int i=0; i<=k;i++)
                System.out.println("Patern "+pattern+" found in string "
                + input +" with group "+ i + " is " +
                                                        m.group(i));

        }
    }
}
```

Output:

```
Total group count= 3
```

```
Patern (.*)(\d{6})(.*) found in string abdpxy 100000 with group 0 is abdpxy
100000
Patern (.*)(\d{6})(.*) found in string abdpxy 100000 with group 1 is abdpxy
Patern (.*)(\d{6})(.*) found in string abdpxy 100000 with group 2 is 100000
Patern (.*)(\d{6})(.*) found in string abdpxy 100000 with group 3 is
```

Example: Write a java program to display the formatted phone number.

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

public class Regex1 {

        public static void main(String[] args) {

                String regex = "\\b(\\d{3})(\\d{3})(\\d{4})\\b";
                Pattern p = Pattern.compile(regex);

                String source = "1234567890, 12345, and 9876543210";
                Matcher m = p.matcher(source);

                int k = m.groupCount();
                System.out.println("Total group count= "+ k);

                while (m.find()) {
                        System.out.println("Phone: " + m.group() + ",
                        Formatted Phone: (" + m.group(1) + ") " + m.group(2) + "-" +
                        m.group(3));
                }
        }

}
```

Output:

```
Total group count= 3
Phone: 1234567890, Formatted Phone: (123) 456-7890
Phone: 9876543210, Formatted Phone: (987) 654-3210
```

Example: If the input is: ***Adams, John Quincy***
and want to get output: ***John Quincy Adams***

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

public class Regex1 {

	public static void main(String[] args) {

		String inputLine = "Adams, John Quincy";
		Pattern r = Pattern.compile("(.*), (.*)");
		Matcher m = r.matcher(inputLine);

		if (m.matches())
			System.out.println(m.group(2) + ' ' + m.group(1));
	}
}
```

Output:

```
John Quincy Adams
```

Replacing the Matched Text

- Replace the text that the regex matched without changing the other text before or after it.

- **replaceAll(newString)**

  - Replaces all occurrences that matched with the new string.

Example:

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

public class Regex1 {

	public static void main(String[] args) {

		String patt = "\\bfavor\\b";
```

```
            String input = "Do me a favor? Fetch my favorite. favor me.";
            System.out.println("Input: " + input);

            Pattern r = Pattern.compile(patt);
            Matcher m = r.matcher(input);
            System.out.println("After Replace: " + m.replaceAll("help"));
        }
}
```

Output:

```
Input: Do me a favor? Fetch my favorite. favor me.
After Replace: Do me a help? Fetch my favorite. help me.
```

Another replacement technique is as follows:

- **appendReplacement(StringBuffer, newString)**
    - Copies up to before the first match, plus the given newString.
- **appendTail(StringBuffer)**
    - Appends text after the last match (normally used after *appendReplacement*).

Example:

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

public class Regex1 {

        public static void main(String[] args) {
           String patt = "favor";
            String input = "Do me a favor? Fetch my favorite. favor me.";
            System.out.println("Input: " + input);

            Pattern r = Pattern.compile(patt);
            Matcher m = r.matcher(input);
```

```
		StringBuffer sb = new StringBuffer();
		System.out.print("Append methods: ");

		while (m.find()) {
			// Copy to before first match, plus the word "help"
			m.appendReplacement(sb, "help");
		}
		m.appendTail(sb); // copy remainder
		System.out.println(sb.toString());
	}
}
```

Output:

```
Input: Do me a favor? Fetch my favorite. favor me.
Append methods: Do me a help? Fetch my favorite. help me.
```

Java Pattern compile() Method

- The compile() method of Pattern class is used to compile the given regular expression passed as the string.
- It used to match text or expression against a regular expression more than one time.
- Syntax:
  - Pattern.compile(String regex)
    - Return compiled version of a regular expression into the pattern.
  - Pattern.compile(String regex, flag)
    - Return compiled version of a regular expression into the pattern with the given flag.
  - Here

    regex - The expression that we want to compile.

flags - Match flags, a bit mask that may includes, CASE_INSENSITIVE, MULTILINE, DOTALL, UNICODE_CASE, CANON_EQ, UNIX_LINES, LITERAL, UNICODE_CHARACTER_CLASS and COMMENTS.

- If more than one value is needed, they can be added together using the bitwise or operator | .

CANON_EQ

- Enables so-called “canonical equivalence.” In other words, characters are matched by their base character, so that the character e followed by the “combining character mark” for the acute accent (́) can be matched either by the composite character é or the letter e followed by the character mark for the accent.

Example:

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

public class PatternCompileFlags {

        public static void main(String[] args) {

                String pattStr = "\u00e9gal";

                String[] input = { "\u00e9gal", "e\u0301gal", "e\u02cagal",
                                                        "e'gal","e\u00b4gal"};

                System.out.println("Pattern: "+ pattStr);

                System.out.print("Input: ");

                for (int i = 0; i < input.length; i++){

                        System.out.print(" " + input[i]);
```

```
            }
            System.out.println(" ");
            Pattern pattern=Pattern.compile(pattStr,
                                            Pattern.CANON_EQ);
            for (int i = 0; i < input.length; i++){
                if(pattern.matcher(input[i]).matches()){
                        System.out.println(pattStr + "matches input
                                        " + input[i]);
                }
                else{
                        System.out.println(pattStr + "does not match
                                    input " + input[i]);
                }
            }
        }
}
```

Output:
Pattern: égal
Input: égal égal e´gal e'gal e´gal

```
égal matches input égal

égal matches input égal

égal does not match input e´gal

égal does not match input e'gal

égal does not match input e´gal
```

CASE_INSENSITIVE and UNICODE_CASE

- **CASE_INSENSITIV:** Compile the Pattern passing in the flags argument Pattern.CASE_INSENSITIVE to indicate that matching should be case-independent (ignore differences in case).

- **UNICODE_CASE:** Enables Unicode-aware case folding. If your code might run in different locales then you should add Pattern.UNICODE_CASE.

Example:

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

public class PatternCompileFlags {

   public static void main(String[] args) {

           String regex = "Pqr";

           String input = "abc Pqr xyz pqr";

           Pattern  p=Pattern.compile(regex, Pattern.CASE_INSENSITIVE
                                            | Pattern.UNICODE_CASE);

          Matcher m=p.matcher(input);

           While (m.find()) {

                 System.out.println("The " + p +"        Pattern found from " +
                                                  m.start() + " to " + (m.end()-1));

          }

     }
}
```

Output:

```
The Pqr Pattern found from 4 to 6
The Pqr Pattern found from 12 to 14
```

MULTILINE, UNIX_LINES and DOTALL

- **MULTILINE**
  - Specifies multiline mode, which makes newlines match as beginning-of-line and end-of-line (\^ and $).

- **DOTALL**
  - Allows dot (.) to match any regular character or the newline, not just any regular character other than newline.

- **UNIX_LINES**
  - Makes \n the only valid “newline” sequence for MULTILINE mode

Example:

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

public class PatternCompileFlags {

    public static void main(String[] args) {

        String input = "I dream of engines\nmore        engines, all day long";
        System.out.println("INPUT: " + input);

        System.out.println();

        String[] patt = {"engines.more engines", "ines\nmore", "engines$"};

        System.out.println("PATTERN: ");

        for (int i = 0; i < patt.length; i++) {

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

        }

        System.out.println();

        for (int i = 0; i < patt.length; i++) {

            System.out.println("PATTERN: " + patt[i]);
            boolean found;
            Pattern p1l = Pattern.compile(patt[i]);
            found = p1l.matcher(input).find();
            System.out.println("DEFAULT match: " + found);
            Pattern pml =  Pattern.compile(patt[i], Pattern.DOTALL |
                                          Pattern.MULTILINE);
            //Pattern pml = Pattern.compile(patt[i], Pattern.DOTALL |
                                              Pattern.UNIX_LINES);
             found = pml.matcher(input).find();
             System.out.println("MultiLine match: " + found);
             System.out.println();
            }
    }
}
```

Output:

INPUT: I dream of engines
more engines, all day long

PATTERN:

engines.more engines

ines

more

engines$

PATTERN: engines.more engines

DEFAULT match: false

MultiLine match: true

PATTERN: ines

more

DEFAULT match: true

MultiLine match: true

PATTERN: engines$

DEFAULT match: false

MultiLine match: true

COMMENTS

- Causes whitespace and comments (from # to end-of-line) to be ignored in the pattern.

Example:

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

public class PatternCompileFlags {

        public static void main(String[] args) {

                String regex = "Pqr #HI I AM HERE.";

                Pattern  p =
                Pattern.compile(regex, Pattern.CASE_INSENSITIVE
                                                    | Pattern.COMMENTS);

                String input = "abc Pqr xyz pqr";

                Matcher m=p.matcher(input);
        while(m.find()) {
            System.out.println("The " + p + "       Pattern found from "
                                        + m.start() + " to " + (m.end()-1));
        }
    }
}
```

Output:

```
The Pqr #HI I AM HERE. Pattern found from 4 to 6
The Pqr #HI I AM HERE. Pattern found from 12 t0 14
```

1 Write a program in Java to remove whitespaces from a string.

```
public class RegexExercise {
    public static void main(String[] args) {
        String str = "ITER Is My College";
        System.out.println("Input String: " + str);
        System.out.println();
        //1st way (using built in method)
        String str1 = str.replaceAll("\\s", "");
        System.out.println("String After Remove Spacce: " + str1);
       System.out.println();
      //2nd way  // using StringBufer() method
       char[] strArray = str.toCharArray();
       StringBuffer stringBuffer = new StringBuffer();
```

```
                for (int i = 0; i < strArray.length; i++) {

                        if ((strArray[i] != ' ') && (strArray[i] != '\t'))
                  {

                                stringBuffer.append(strArray[i]);

                        }

                  }

                String str2 = stringBuffer.toString();

                  System.out.println("String After Remove Spacce: " + str2);

        }

}
```

Output:

```
Input String: ITER Is My College

String After Remove Spacce: ITERIsMyCollege

String After Remove Spacce: ITERIsMyCollege
```

2 Write a program in Java to check valid mobile number.

Mobile Number validation criteria:

- The first digit should contain number between 7 to 9.
- The rest 9 digit can contain any number between 0 to 9.
- The mobile number can have 11 digits also by including 0 at the starting.

- The mobile number can be of 12 digits also by including 91 at the starting

The number which satisfies the above criteria, is a valid mobile Number.

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

class MobileNumberVelidation {

    public static void main(String[] args) {

                String regex = "(0|(91))?[789][0-9]{9}";
                Pattern ptr = Pattern.compile(regex);
                String mobileno = "9439342133";      //valid.
                //String mobileno = "09439342133";   //valid.
                //String mobileno = "919439342133"; //valid.
                //String mobileno = "913439342133";   //not valid.
                Matcher m = ptr.matcher(mobileno);
                if(m.matches())
                {
                        System.out.println("The Mobile number " + mobileno +
                                                        " is valid.");
                }
                else
                {
                        System.out.println("The Mobile number " + mobileno +
                                                     " is not valid.");
                }
        }
}
```

Output :

```
The Mobile number 9439342133 is valid.
```

3 Write a program in Java to check valid email

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

public class RegexExercise {

        public static void main(String[] args) {

                String emailRegex = "^([a-zA-Z0-9_\\-\\.]
                +)@([a-zA-Z0-9_\\-\\.]+)\\.([a-zA-Z]{2,5})$";

                Pattern pat = Pattern.compile(emailRegex);

                String email = "contribute@gmail.co.in";

                if (pat.matcher(email).matches())

                        System.out.print("Yes");

                else

                        System.out.print("No");

        }

}
```

Output:

```
yes
```

4. Write a program to extract maximum numeric value from a given string. For example: this is âĂIJThere is 60 students in csed section, 40 in cseb, and 55 in cseaâĂİ.

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

public class RegexExercise {
    public static void main(String[] args) {
        String str = "this is âĂIJThere is 60 students in csed section,
                                    40 in cseb, and 55 in cseaâĂİ";
        String regex = "\\d+";
        Pattern p = Pattern.compile(regex);
        Matcher m = p.matcher(str);
        int MAX = 0;
        while(m.find()) {
            int num = Integer.parseInt(m.group());
            if(num > MAX)
                MAX = num;
        }
        System.out.println(MAX);
    }
}
```

Output:

```
60
```

5 Write a program to demonstrate the working of splitting a text by a given pattern. The given input is “CSE1ECE2EEE3CIVIL”.The output of the program is look like below:

CSE

ECE

EEE

CIVIL

Use the split () and case controlling flags to solve this.

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

public class RegexExercise {

        public static void main(String[] args) {

                String s1 = "CSE1ECE2EEE3CIVIL";
                String regex1 = "\\d";


                String[] arrOfStr = s1.split(regex1, 5);

                        for (String a : arrOfStr)
                                System.out.println(a);
        }
}
```

Output:

```
CSE
ECE
EEE
CIVIL
```

6 Write a program to get the first letter of each word in a string using regex in Java. For example: the input string is "This is CSE Students" and output of the program is: TiCS.

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

public class RegexExercise {

        public static void main(String[] args) {
                String s1 = "This is CSE Students";
                Pattern p = Pattern.compile("\\b[a-zA-Z]");
                Matcher m = p.matcher(s1);
                while (m.find())
                        System.out.print(m.group());
        }
}
```

Output:

```
TiCS
```

7 Write a program to demonstrate the differences of various quantifiers e.g. Greedy Quantifiers VS Reluctant Quantifiers VS Possessive Quantifiers.

Regular Expressions in Java

Quantifiers allow user to specify the number of occurrences to match against.

Below are some commonly used quantifiers in Java.

X*	Zero or more occurrences of X
X?	Zero or One occurrences of X
X+	One or More occurrences of X
X{n}	Exactly n occurrences of X
X{n, }	At-least n occurrences of X
X{n, m}	Count of occurrences of X is from n to m

The above quantifiers can be made Greedy, Reluctant and Possessive.

Greedy quantifier (Default)

By default, quantifiers are Greedy. Greedy quantifiers try to match the longest text that matches given pattern. Greedy quantifiers work by first reading the entire string before trying any match. If the entire text doesn't match, remove last character and try again, repeating the process until a match is found.

```
// Java program to demonstrate Greedy Quantifiers
import java.util.regex.Matcher;
import java.util.regex.Pattern;
 class Test
{
    public static void main(String[] args)
    {
        // Making an instance of Pattern class. By default quantifier "+" is
          //Greedy
        Pattern p = Pattern.compile("g+");
       // Making an instance of Matcher class
        Matcher m = p.matcher("ggg");
       while (m.find())
            System.out.println("Pattern found from " + m.start() +
                                                        " to " + (m.end()-1));
       }
}
```

Output :

```
Pattern found from 0 to 2
```

Explanation : The pattern **g+** means one or more occurrences of **g**. Text is **ggg**. The greedy matcher would match the longest text even if parts of matching text also match. In this example, **g** and **gg** also match, but the greedy matcher produces **ggg**.

Reluctant quantifier (Appending a **?** after quantifier)

This quantifier uses the approach that is opposite of greedy quantifiers. It starts from first character and processes one character at a time.

```
// Java program to demonstrate Reluctant Quantifiers
import java.util.regex.Matcher;
import java.util.regex.Pattern;
class Test {
    public static void main(String[] args)
    {
        // Making an instance of Pattern class. Here "+" is a Reluctant
          //quantifier because a "?' is appended after it.
        Pattern p = Pattern.compile("g+?");
        // Making an instance of Matcher class
        Matcher m = p.matcher("ggg");
           while (m.find())
```

```
            System.out.println("Pattern found from " + m.start() + " to " +
                                                      (m.end()-1));


    }
}
```

Output :

```
Pattern found from 0 to 0

Pattern found from 1 to 1

Pattern found from 2 to 2
```

Explanation : Since the quantifier is reluctant, it matches the shortest part of test with pattern. It processes one character at a time.

Possessive quantifier (Appending a **+** after quantifier)

This quantifier matches as many characters as it can like greedy quantifier. But if the entire string doesn't match, then it doesn't try removing characters from end.

```
// Java program to demonstrate Possessive Quantifiers
import java.util.regex.Matcher;
import java.util.regex.Pattern;
 class Test
{
    public static void main(String[] args)
    {
        // Making an instance of Pattern class.Here "+" is a Possessive     quantifier
because a "+' is appended after it.

        Pattern p = Pattern.compile("g++");

       // Making an instance of Matcher class

        Matcher m = p.matcher("ggg");

        while (m.find())

            System.out.println("Pattern found from " + m.start() + " to " +
                                                                (m.end()-1));

    }

}
```

Output :

```
Pattern found from 0 to 2
```

Explanation: We get the same output as Greedy because whole text matches the pattern.

Below is an example to show difference between Greedy and Possessive Quantifiers.

```
// Java program to demonstrate difference between Possessive and

// Greedy Quantifiers

import java.util.regex.Matcher;

import java.util.regex.Pattern;

class Test {

    public static void main(String[] args)

    {
        // Create a pattern with Greedy quantifier
        Pattern pg = Pattern.compile("g+g");
        // Create same pattern with possessive quantifier
        Pattern pp = Pattern.compile("g++g");
       System.out.println("Using Greedy Quantifier");
        Matcher mg = pg.matcher("ggg");
        while (mg.find())
                System.out.println("Pattern found from " + mg.start() + " to "
                                                        + (mg.end()-1));
        System.out.println("\nUsing Possessive Quantifier");
        Matcher mp = pp.matcher("ggg");
        while (mp.find())
                System.out.println("Pattern found from " + mp.start() + " to "
                                                        + (mp.end()-1));
        }
}
```

Output :

```
Using Greedy Quantifier

Pattern found from 0 to 2
```

Using Possessive Quantifier

In the above example, since first quantifier is greedy, **g+** matches with whole string. If we match **g+** with whole string, **g+g** doesn’t match, the Greedy quantifier, removes last character, matches **gg** with **g+** and finds a match.

In Possessive quantifier, we start like Greedy. **g+** matches whole string, but matching **g+** with whole string doesn’t match **g+g** with **ggg**. Unlike Greedy, since quantifier is possessive, we stop at this point.