Often you need to write the code to validate user input such as to check whether the input is a number, a string with all lowercase letters, or a social security number. How do you write this type of code? A simple and effective way to accomplish this task is to use the regular expression.

A regular expression (abbreviated regex) is a string that describes a pattern for matching a set of strings. Regular expression is a powerful tool for string manipulations. You can use regular expressions for matching, replacing, and splitting strings.

1 Matching Strings

Let us begin with the matches method in the String class. At first glance, the matches method is very similar to the equals method. For example, the following two statements both evaluate to true.

```
"Java".matches("Java");
"Java".equals("Java");
```

However, the matches method is more powerful. It can match not only a fixed string, but also a set of strings that follow a pattern. For example, the following statements all evaluate to true.

```
"Java is fun".matches("Java.*")
"Java is cool".matches("Java.*")
"Java is powerful".matches("Java.*")
```

"Java.*" in the preceding statements is a regular expression. It describes a string pattern that begins with Java followed by any zero or more characters. Here, the substring .* matches any zero or more characters.

2 Regular Expression Syntax

A regular expression consists of literal characters and special symbols. Table 1 lists some frequently used syntax for regular expressions.

Table 1: Frequently Used Regular Expressions
### Regular Expression

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Matches</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>a specified character x</td>
<td>Java matches Java</td>
</tr>
<tr>
<td>.</td>
<td>any single character .</td>
<td>Java matches J.a</td>
</tr>
<tr>
<td>(ab</td>
<td>cd)</td>
<td>ab or cd (ab</td>
</tr>
<tr>
<td>[abc]</td>
<td>a, b, or c [abc]</td>
<td>Java matches Ja</td>
</tr>
<tr>
<td>[^abc]</td>
<td>any character except a, b, or c [^abc]</td>
<td>Java matches &quot;ars</td>
</tr>
<tr>
<td>[a-z]</td>
<td>a through z [a-z]</td>
<td>Java matches [A-M]av[a-d]</td>
</tr>
<tr>
<td>(^a-z)</td>
<td>any character except a through z (^a-z)</td>
<td>Java matches Java</td>
</tr>
<tr>
<td>[a-e[m-p]]</td>
<td>a through e or m through p [a-e[m-p]]</td>
<td>ten matches t(en</td>
</tr>
<tr>
<td>[a-e&amp;[c-p]]</td>
<td>intersection of a-e with c-p [a-e&amp;[c-p]]</td>
<td>Java matches [A-P&amp;[I-M]av[a-d]</td>
</tr>
<tr>
<td>\d</td>
<td>a digit, same as [0-9] \d</td>
<td>Java matches &quot;Java[\d]&quot;</td>
</tr>
<tr>
<td>\D</td>
<td>a non-digit \D</td>
<td>$Java matches &quot;\D$$Java&quot;</td>
</tr>
<tr>
<td>\w</td>
<td>a word character \w</td>
<td>Java matches &quot;\w\w\w\wJava&quot;</td>
</tr>
<tr>
<td>\W</td>
<td>a non-word character \W</td>
<td>$Java matches &quot;\W\W\W\WJava&quot;</td>
</tr>
<tr>
<td>\s</td>
<td>a whitespace character \s</td>
<td>&quot;Java 2&quot; matches &quot;Java\s2&quot;</td>
</tr>
<tr>
<td>\S</td>
<td>a non-whitespace char \S</td>
<td>Java matches &quot;\S\SJava&quot;</td>
</tr>
<tr>
<td>p*</td>
<td>zero or more occurrences of pattern p p*</td>
<td>Java matches &quot;a*&quot;</td>
</tr>
<tr>
<td>p+</td>
<td>one or more occurrences of pattern p p+</td>
<td>bb matches &quot;a*&quot;</td>
</tr>
<tr>
<td>p?</td>
<td>zero or one occurrence of pattern p p?</td>
<td>Java matches &quot;J?Java&quot;</td>
</tr>
<tr>
<td>p{n}</td>
<td>exactly n occurrences of pattern p p{n}</td>
<td>Java matches &quot;a{1}&quot;</td>
</tr>
<tr>
<td>p{n,}</td>
<td>at least n occurrences of pattern p p{n,}</td>
<td>Java matches &quot;a{1,}&quot;</td>
</tr>
<tr>
<td>p{n,m}</td>
<td>between n and m occurrences (inclusive) p{n,m}</td>
<td>Java matches &quot;a{1,9}&quot;</td>
</tr>
</tbody>
</table>

**NOTE**

Backslash is a special character that starts an escape sequence in a string. So you need to use "\\d" in Java to represent \d.

**NOTE**

Recall that a whitespace (or a whitespace character) is any character which does not display itself but does take up space. The characters ' ', '	', '"', '', '"f' are whitespace characters. So \s is the same as [ \t\n\r\f], and \S is the same as [^ \t\n\r\f\v].

**NOTE**

A word character is any letter, digit, or the underscore character. So \w is the same as [a-z[A-Z][0-9]_] or simply [a-zA-Z0-9_], and \W is the same as [^a-zA-Z0-9_].

**NOTE**
The last six entries *, +, ?, {n}, {n,}, and {n, m} in Table 1 are called quantifiers that specify how many times the pattern before a quantifier may repeat. For example, A* matches zero or more A’s, A+ matches one or more A’s, A? matches zero or one A’s, A{3} matches exactly AAA, A{3,} matches at least three A’s, and A{3,6} matches between 3 and 6 A’s. * is the same as {0,}, + is the same as {1,}, and ? is the same as {0,1}.

CAUTION
Do not use spaces in the repeat quantifiers. For example, A{3,6} cannot be written as A{3,6} with a space after the comma.

NOTE
You may use parentheses to group patterns. For example, (ab){3} matches ababab, but ab{3} matches abbb.

Let us use several examples to demonstrate how to construct regular expressions.

**Example 1:** The pattern for social security numbers is xxx-xx-xxxx, where x is a digit. A regular expression for social security numbers can be described as

```
[\d]{3}-[\d]{2}-[\d]{4}
```

For example,

- "111-22-3333".matches("[\d]{3}-[\d]{2}-[\d]{4}")) returns true.
- "11-22-3333".matches("[\d]{3}-[\d]{2}-[\d]{4}")) returns false.

**Example 2:** An even number ends with digits 0, 2, 4, 6, or 8. The pattern for even numbers can be described as

```
[\d]*[02468]
```

For example,

- "123".matches("[\d]*[02468]")) returns false.
- "122".matches("[\d]*[02468]")) returns true.

**Example 3:** The pattern for telephone numbers is (xxx) xxx-xxxx, where x is a digit and the first digit cannot be zero. A regular expression for telephone numbers can be described as

```
\((\[1-9]\[\d]{2}\) \[\d]{3}-[\d]{4}
```
Note that the parentheses symbols ( and ) are special characters in a regular expression for grouping patterns. To represent a literal ( or ) in a regular expression, you have to use \( and \). For example,

- "(912) 921-2728".matches("\(\[(1-9)\[d\](2)\]\) \(\d\)(3)-(\d\)(4)\)") returns true.
- "921-2728".matches("\(\[(1-9)\[d\](2)\]\) \(\d\)(3)-(\d\)(4)\)") returns false.

**Example 4:** Suppose the last name consists of at most 25 letters and the first letter is in uppercase. The pattern for a last name can be described as

\[A-Z][a-zA-Z]{1,24}\]

Note that you cannot have arbitrary whitespace in a regular expression. For example, \[A-Z][a-zA-Z]{1,24}\] would be wrong.

For example,

- "Smith".matches("\[A-Z][a-zA-Z]{1,24}\)") returns true.
- "Jones123".matches("\[A-Z][a-zA-Z]{1,24}\)") returns false.

**Example 5:** Java identifiers are defined in §2.3, "Identifiers."

- An identifier must start with a letter, an underscore (_), or a dollar sign ($). It cannot start with a digit.
- An identifier is a sequence of characters that consists of letters, digits, underscores (_), and dollar signs ($).

The pattern for identifiers can be described as

\[a-zA-Z_\]\\w\]*

**Example 6:** What strings are matched by the regular expression "Welcome to (Java|HTML)"? The answer is Welcome to Java or Welcome to HTML.

**Example 7:** What strings are matched by the regular expression ".*"? The answer is any string.

3 Replacing and Splitting Strings
The matches method in the String class returns true if the string matches the regular expression. The String class also contains the replaceAll, replaceFirst, and split methods for replacing and splitting strings, as shown in Figure 1.
Figure 1
The *String* class contains the methods for matching, replacing, and splitting strings using regular expressions.

The *replaceAll* method replaces all matching substring and the *replaceFirst* method replaces the first matching substring. For example, the following code

```
System.out.println("Java Java Java".replaceAll("v\w", "wi"));
```

displays

```
Jawi Jawi Jawi
```

The following code

```
System.out.println("Java Java Java".replaceFirst("v\w", "wi"));
```

displays

```
Jawi Java Java
```

There are two overloaded *split* methods. The *split(regex)* method splits a string into substrings delimited by the matches. For example, the following statement

```
String[] tokens = "Java1HTML2Perl".split("\d");
```


In the *split(regex, limit)* method, the `limit` parameter determines how many times the pattern is matched. If `limit <= 0`, *split(regex, limit)* is same as *split(regex)*. If `limit > 0`, the pattern is matched at most `limit - 1` times. Here are some examples:

```
"Java1HTML2Perl".split("\d", 0);  // splits into Java, HTML, Perl
"Java1HTML2Perl".split("\d", 1);  // splits into Java1HTML2Perl
"Java1HTML2Perl".split("\d", 2);  // splits into Java, HTML2Perl
"Java1HTML2Perl".split("\d", 3);  // splits into Java, HTML, Perl
"Java1HTML2Perl".split("\d", 4);  // splits into Java, HTML, Perl
"Java1HTML2Perl".split("\d", 5);  // splits into Java, HTML, Perl
```
NOTE:
By default, all the quantifiers are greedy. This means that they will match as many occurrences as possible. For example, the following statement displays \texttt{JRvaa}, since the first match is \texttt{aaa}.

\begin{verbatim}
System.out.println("Jaaavaa".replaceFirst("a+", "R"));
\end{verbatim}

You can change a qualifier’s default behavior by appending a question mark (?) after it. The quantifier becomes reluctant, which means that it will match as few occurrences as possible. For example, the following statement displays \texttt{JRaavaa}, since the first match is \texttt{a}.

\begin{verbatim}
System.out.println("Jaaavaa".replaceFirst("a+?", "R"));
\end{verbatim}