



Java Programming

7-2

Use Regular Expressions



Objectives

This lesson covers the following topics:

- Use regular expressions
- Use regular expressions to:
 - Search Strings
 - Parse Strings
 - Replace Strings



Regular Expressions

- A regular expression is a character or a sequence of characters that represent a String or multiple Strings.
- Regular expressions:
 - Are part of the `java.util.regex` package, thus any time regular expressions are used in your program you must import this package.
 - Syntax is different than what you are used to but allows for quicker, easier searching, parsing, and replacing of characters in a String.



String.matches(String regex)

- The String class contains a method named matches(String regex) that returns true if a String matches the given regular expression.
- This is similar to the String method equals(String str).
- The difference is that comparing the String to a regular expression allows variability.
- For example, how would you write code that returns true if the String animal is “cat” or “dog” and returns false otherwise?

Equals Versus Matches

- A standard answer may look something like this:

```
if (animal.equals("cat"))  
    return true;  
else if (animal.equals("dog"))  
    return true;  
return false;
```

- An answer using regular expressions would look something like this:

```
return animal.matches("cat|dog");
```

- The second solution is much shorter. The regular expression symbol | allows for the method matches to check if animal is equal to "cat" or "dog" and return true accordingly.

Equals Versus Matches Example

```
package regExp;

import java.util.regex.*;

public class RegularExpDemo {

    public static void main(String[] args) {
        if(getAnimal("cat"))
            System.out.println("This is a Valid Animal");
        else
            System.out.println("This is not a Valid Animal");
        //endif
    } //end of method main

    public static boolean getAnimal(String animal){
        return animal.matches("cat|dog");
    } //end of method getAnimal
} //end of class
```

Square Brackets

- Square brackets are used in regular expression to allow for character variability.
- If you wanted to return true if animal is equal to “dog” or “Dog”, but not “dOg”, using equalsIgnoreCase would not work and using equals would take time and multiple lines.
- If you use regular expression, this task can be done in one line as follows.
- This code tests if animal matches “Cat” or “cat” or “Dog” or “dog” and returns true if it does.

```
return animal.matches("[Cc]at|[Dd]og");
```


Include Any Range of Characters

- Square brackets aren't restricted to two character options.
- They can be combined with a hyphen to include any range of characters.
- For example, you are writing code to create a rhyming game and you want to see if a String word rhymes with mouse.
- The definition of a rhyming word is a word that contains all the same letters except the first letter may be any letter of the alphabet.

Include Any Range of Characters

Your first attempt at coding may look like this:

```
if(word.length()==5)
    if(word.substring(1,5).equals("ouse"))
        return true;
return false;
```

Using Square Brackets and a Hyphen

- A shorter, more generic way to complete the same task is to use square brackets and a hyphen (regular expression) as shown below.

```
return word.matches("[a-z]ouse");
```

- This code returns true if word begins with any lower case letter and ends in "ouse".
- To include upper case characters we would write:

```
return word.matches("[a-zA-Z]ouse");
```

Using Square Brackets and a Hyphen Example

```
import java.util.Scanner;

public class RegularExp2 {
    public static void main(String[] args) {
        String animal;
        animal = getAnimal();
        if(rhymningAnimal(animal))
            System.out.println("This animal rhymes with mouse!");
        else
            System.out.println("This animal doesn't rhyme!");
        //endif
    } //end method main

    private static boolean rhymningAnimal(String animal){
        return animal.matches("[a-zA-Z]ouse");
    } //end method rhymningAnimal

    private static String getAnimal(){
        String animal;
        Scanner in = new Scanner(System.in);
        System.out.print("Please enter the name of the animal: ");
        animal = in.next();
        in.close();
        return animal;
    } //end method getAnimal
} //end class RegularExp2
```

Using Square Brackets and a Hyphen

To allow the first character to be any number or a space in addition to a lower or upper case character, simply add “ 0-9” inside the brackets (note the space before 0).

```
return word.matches("[ 0-9a-zA-Z]ouse");
```



The Dot

- The dot (.) is a representation for any character in regular expressions.
- For example, you are writing a decoder for a top secret company and you think that you have cracked the code.
- You need to see if String element consists of a number followed by any other single character.

The Dot

- This task is done easily with use of the dot as shown below.
- This code returns true if element consists of a number followed by any character.
- The dot matches any character.

```
return element.matches("[0-9].");
```

Repetition Operators

A repetition operator is any symbol in regular expressions that indicates the number of times a specified character appears in a matching String.

Repetition Operator	Definition	Sample Code	Code Meaning
*	0 or more occurrences	<pre>return str.matches("A*");</pre>	Returns true if str consists of zero or more A's but no other letter.
?	0 or 1 occurrence	<pre>return str.matches("A?");</pre>	Returns true if str is "" or "A".
+	1 or more occurrences	<pre>return str.matches("A+");</pre>	Returns true if str is 1 or more A's in a sequence.

More Repetition Operators

A repetition operator is any symbol in regular expressions that indicates the number of times a specified character appears in a matching String.

Repetition Operator	Definition	Sample Code	Code Meaning
{x}	X occurrences	<code>return str.matches("A{7}");</code>	Returns true if str is a sequence of 7 A's.
{x,y}	Between x & y occurrences	<code>return str.matches("A{7,9}");</code>	Returns true if str is a sequence of 7, 8, or 9 A's.
{x,}	X or more occurrences	<code>Return str.matches("A{5,}");</code>	Returns true if str is a sequence of 5 or more A's.

Combining Repetition Operators Example 1

In the code below:

- The dot represents any character.
- The asterisk represents any number of occurrences of the character preceding it.
- The “.” means any number of any characters in a sequence will return true.

```
return str.matches(“.*”);
```

Combining Repetition Operators Example 2

- If the code below returns true, str must be a sequence of 10 digits (between 0 and 5) and may have 0 or 1 characters preceding the sequence.
- Remember, all symbols of regular expressions may be combined with each other, as shown below, and with standard characters.

```
return str.matches(".?[0-5]{10}");
```

Combining Repetition Operators Example

```
import java.util.Scanner;
public class RegEx3 {
    public static void main(String[] args) {
        String ssn;
        ssn = getSsn();
        if(validSsn(ssn))
            System.out.println("This ssn is valid!");
        else
            System.out.println("This ssn is not valid! must be in the format of (999- 99-9999)");
        //endif
    } //end method main

    private static boolean validSsn(String ssn){
        return ssn.matches("[0-9]{3}-[0-9]{2}-[0-9]{4}");
    } //end method rhymningAnimal

    private static String getSsn(){
        String ssn;
        Scanner in = new Scanner(System.in);
        System.out.print("Please enter your Social Security Number: ");
        ssn = in.next();
        in.close();
        return ssn;
    } //end method getSsn
} //end class RegEx3
```

Pattern

- A Pattern is a class in the java.util.regex package that stores the format of the regular expression.
- For example, to initialize a Pattern of characters as defined by the regular expression “[A-F]{5,}.*” you would write the following code:

```
Pattern p = Pattern.compile("[A-F]{5,}.*");
```

- The compile method returns a Pattern as defined by the regular expression given in the parameter.

Matcher

- A matcher is a class in the `java.util.regex` package that stores a possible match between the `Pattern` and a `String`.
- A `Matcher` is initialized as follows:

```
Matcher match = patternName.matcher(StringName);
```

- The `matcher` method returns a `Matcher` object.
- The following code returns `true` if the regular expression given in the `Pattern` `patternName` declaration matches the `String` `StringName`.

```
return match.matches();
```

Matcher: Putting it All Together

To put it all together, we have:

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

public class PatternTest{

    public static void main(String[] args) {
        Pattern p = Pattern.compile("[A-F]{5,}.*");
        String str="AAAAAhhhh";
        boolean matched;

        matched = isMatch(str, p);
        System.out.println(matched);
    } //end of method main

    private static boolean isMatch(String str, Pattern p){
        Matcher match = p.matcher(str);
        return match.matches();
    } //end of method isMatch
} //end of class PatternTest
```

Benefits to Using Pattern and Matcher

- This seems like a very complex way of completing the same task as the String method matches.
- Although that may be true, there are benefits to using a Pattern and Matcher such as:
- Capturing groups of Strings and pulling them out, allowing to keep specific formats for dates or other specific formats without having to create special classes for them.
- Matches has a find() method that allows for detection of multiple instances of a pattern within the same String.



Regular Expressions and Groups

- Segments of regular expressions can be grouped using parentheses, opening the group with “(“ and closing it with “)”.
- These groups can later be accessed with the Matcher method `group(groupNumber)`.
- For example, consider reading in a sequence of dates, Strings in the format “DD/MM/YYYY”, and printing out each date in the format “MM/DD/YYYY”.
- Using groups would make this task quite simple.

Regular Expressions and Example

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

public class RegExpressionsPractice {
    public static void main(String[] args) {
        Pattern dateP;
        dateP = Pattern.compile("([0-9]{2})/([0-9]{2})/([0-9]{4})");
        Scanner in = new Scanner(System.in);
        System.out.print("Enter a Date (dd/mm/yyyy): ");
        String date = in.nextLine();
        while(!date.equals("")){
            Matcher dateM = dateP.matcher(date);
            if(dateM.matches()){
                String day = dateM.group(1);
                String month = dateM.group(2);
                String year = dateM.group(3);
                System.out.println("US style date - " + month + "/" + day + "/" + year);
            }//endif
            System.out.print("Enter a Date (dd/mm/yyyy): ");
            date=in.nextLine();
        }//endwhile
        in.close();
    }//end method main
}//end class RegExpressionsPractice
```

Group 1

Group 2

Group 3

Recalls each group of the Matcher.

Group 1 and Group 2 are defined to consist of 2 digits each. Group 3 (the year) is defined to consist of 4 digits. Note: It is still possible to get the whole Matcher by calling group (0).

Matcher.find()

- Matcher's find method will return true if the defined Pattern exists as a Substring of the String of the Matcher.
- For example, if we had a pattern defined by the regular expression “[0-9]”, as long as we give the Matcher a String that contains at least one digit somewhere in the String, calling find() on this Matcher will return true.



Parsing a String with Regular Expressions

- Recall the String method `split()` introduced earlier in the lesson, which splits a String by spaces and returns the split Strings in an array of Strings.
- The `split` method has an optional parameter, a regular expression that describes where the operator wishes to split the String.
- For example, if we wished to split the String at any sequence of one or more digits, we could write something like this:

```
String[] tokens = str.split("[0-9]+");
```

Replacing with Regular Expressions

- There are a few simple options for replacing Substrings using regular expressions.
- The following is the most commonly used method.
- `replaceAll` - For use with Strings, the method:

```
replaceAll("RegularExpression", newSubstring)
```

- `replaceAll` will replace all occurrences of the defined regular expression found in the String with the defined String `newSubstring`.
- Other methods that could be used are `replaceFirst()` and `split()` that can be both be researched through the Java API.

Replacing with Regular Expressions Example

The following example will use a regular expression to remove multiple spaces from a String and replace them with a substring that consists of a single space.

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

public class RegExpressionsReplaceDemo {
    public static void main(String[] args) {
        String str = "help me I have no idea what's going on! ! !";
        str = str.replaceAll(" {2,}", " ");
        System.out.println(str);
    } //end method main
} //end class RegExpressionsReplaceDemo
```

Replacing using a Matcher

- ReplaceAll - For use with a matcher
- This method works the same if called by a Matcher rather than a String. However, it does not require the regular expression.
- It will simply replace any matches of the Pattern you gave it when you initialized the Matcher.
- The method example shown below results in a replacement of all matches identified by Matcher with the String “abc”.

```
MatcherName.replaceAll("abc");
```

Replacing using a Matcher Example

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

public class RegExpressionsMatcher {
    public static void main(String[] args) {
        //create the pattern
        Pattern p = Pattern.compile("(J|j)ava");
        //create the initial String
        String str = "Java courses are the best! You have got to love java.";
        //print the contents of the string to screen
        System.out.println(str);
        //initialise the matcher
        Matcher m = p.matcher(str);
        //replace all occurrences of the pattern with the new substring
        str = m.replaceAll("Oracle");
        //print the contents of the string to screen
        System.out.println(str);
    } //end method main
} //end class RegExpressionsMatcher
```


Terminology

Key terms used in this lesson included:

- Regular Expression
- Matcher
- Pattern
- Parsing
- Dot
- Groups
- Square Brackets
- Repetition Operator

Summary

In this lesson, you should have learned how to:

- Use regular expressions
- Use regular expressions to:
 - Search Strings
 - Parse Strings
 - Replace Strings

