
ORACLE
Academy

Java Programming

4-2

Use Regular Expressions

ORACLE
Academy

Objectives

- This lesson covers the following topics:
  - Understand 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, this package has classes to help form and use regular expressions
  - Syntax is different than what you are used to but allows for quicker, easier searching, parsing, and replacing of characters in a String

Regular expression are used to ensure that Strings contain specific contents and are often used to check correct email format (@ sign is included) on form validation.

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?

The matches method allows you to determine exactly what the String should match against and allows you to specify multiple values.

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!

You should be familiar with the String equals method that compares one String to another by comparing a single argument.
The matches method allows much more flexibility in your code.

Equals Versus Matches Example

1. The regular expression symbol | allows for the method matches to check if animal is equal to “cat” or “dog” and return true accordingly

```
package regularexpressionexample;
public class RegularExpressionExample {
   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
```

Test it by using a value other than cat or dog.

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 multiple lines

2. 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

```
public static boolean getAnimal(String animal){
   return animal.matches("[Cc]at|[Dd]og");
}//end of method getAnimal
```

Update the code in the program.

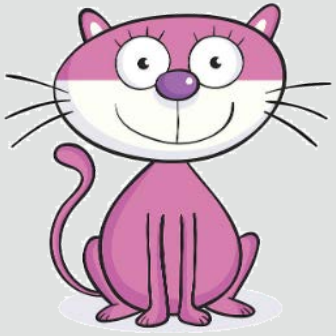
Include Any Range of Characters

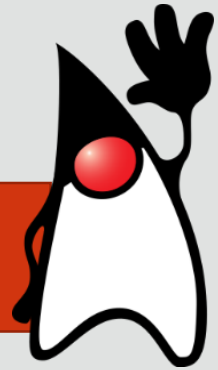
- Square brackets aren't restricted to two-character options
- They can be combined with a hyphen to include any range of characters
- For example, you are writing code to create a rhyming game
  - You want to see if a String word rhymes with cat
  - The definition of a rhyming word is a word that contains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

Include Any Range of Characters

- Your first attempt at coding may look like this:

```
if(word.length()==3)
   if(word.substring(1,3).equals("at"))
      return true;
   //endif
return false;
```





Up until now this is how you would have solved this problem, but regular expressions make it much easier.

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

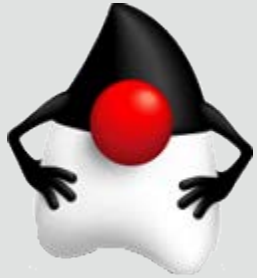
```
return word.matches("[a-z]at");
```

- This code returns true if word begins with any lower-case letter and ends in “ouse”
- To include upper case characters write:

```
return word.matches("[a-zA-Z](at|AT)");
```

This allows specific characters to be upper or lowercase.
Note: Do not use the range a-Z or A-z. These will not work correctly.

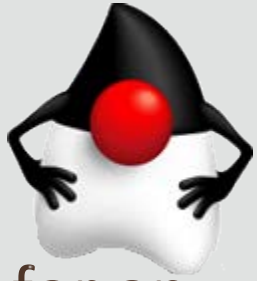
Regular Expressions Task1



a) Create the following base class for the task:

```
package rhyminggame;

import java.util.Scanner;

public class RhymeGame {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      String animal;
      in.close();
   }//end method main

   private static  boolean rhymningAnimal(String animal){
   }//end method rhymningAnimal

   private static String getAnimal(Scanner in){
   }//end method getAnimal
}//end class RyhmeGame
```

Regular Expressions Task1



b) Code the getAnimal() method to ask the user for an animal through the console, use this prompt:

```
Please enter the name of the animal:
```

c) Use a regular expression in the rhymningAnimal() method that will return true if the animal name matches, upper and lower case should be catered for

d) Use the result of rhymningAnimal() in an if statement in main to display either of these messages:

```
This animal rhymes with cat!
```

or

```
This animal doesn't rhyme!
```

Regular Expressions Task1 Suggested Solution

```
package rhyminggame;

import java.util.Scanner;

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

Regular Expressions Task1 Suggested Solution

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

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

}//end class RyhmeGame
```

Using Square Brackets and a Hyphen

- This code allows the first character to be:
  - a space (note the space before 0)
  - any number
  - a lower or upper case character
- The rest of the word must be bcde and is case sensitive

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

Everything inside of the square brackets defines exactly how the first character can be represented.

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 a 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].");
```

The dot is the wildcard operator that represents any single character.

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	`return str.matches("A*");`	Returns true if str consists of **zero or more** A's but no other letter.
?	0 or 1 occurrence	`return str.matches("A?");`	Returns true if str is **""** **or** "A".
+	1 or more occurrences	`return str.matches("A+");`	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	`return str.matches("A{7}");`	Returns true if str is a **sequence of 7** A's.
`{x,y}`	Between x & y occurrences	`return str.matches("A{7,9}");`	Returns true if str is a **sequence of 7, 8, or 9** A's.
`{x,}`	x or more occurrences	`return str.matches("A{5,}");`	Returns true if str is a **sequence of 5** or more A's.

Combining Repetition Operators

- 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(".*");
```

As you can see regular expressions allow for very powerful validation of Strings without having to write very much code.

Combining Repetition Operators

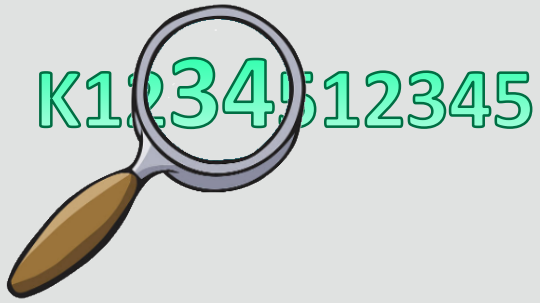
- For the code below to return true, str must be a sequence of 10 digits (between 0 and 5) and may have 0 or 1 characters preceding the sequence
- Remember, all symbols of regular expressions may be combined with each other, as shown below, and with standard characters

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

Valid Code : K1234512345

Invalid Code: K1234567890

Invalid Code: KL1234512345



Repetition Operators Example

- Let's create a program that checks if a valid SSN (United States Social Security Number) has been entered
  - The correct format is three numbers followed by a dash, followed by two numbers, followed by a dash, followed by four numbers
  - Any other input will be treated as being invalid

1. Create a class named SsnCheck

```
public class SsnCheck {
   public static void main(String[] args) {
   }//end method main
}//end class SsnCheck
```

Repetition Operators Example

2. Create the following method that uses a regular expression to enforce the format of an SSN
   - three numbers
   - a dash
   - two numbers
   - a dash
   - four numbers

```
    }//end method main

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

Repetition Operators Example

3. Create the following method that uses a Scanner object to get a value for the SSN from the user

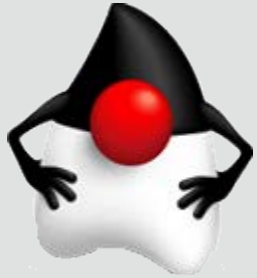
```
 }//end method rhymningAnimal

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

}//end class SsnCheck
```

Repetition Operators Example

4. Update the main method to create the required local variables and call the relevant methods

```
import java.util.Scanner;

public class SsnCheck {
   static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      String ssn;
      ssn = getSsn(in);
      if(validSsn(ssn))
         System.out.println(ssn + " is a valid Social Security Number!");
      else
         System.out.println("This ssn is not valid! must be in the format
                             of (999-99-9999)");
      //endif
      in.close();
}//end method main
```

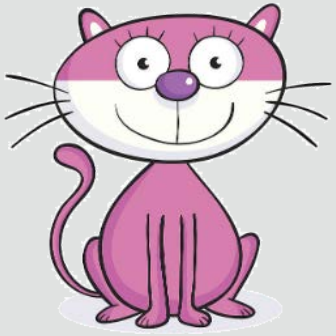
Regular Expressions Quick Task



a) Open the rhymingame project and load the RyhmeGame.java class

b) Update the rhymingAnimal method to allow either 1 or 2 characters to come before the at|AT characters for the rhyme to work

c) This should let animals like gnat now match the rhyming criteria



ORACLE Academy

Regular Expressions Quick Task Suggested Solution

```
public class RyhmeGame {
   public static void main(String[] args) {
      {..code..}
   }//end method main

   private static  boolean rhymningAnimal(String animal){
       return animal.matches("[a-zA-Z]{1,2}(at|AT)");
   }//end method rhymningAnimal

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

}//end class RyhmeGame
```

Pattern

- A Pattern is a class in the java.util.regex package that stores the format of the regular expression

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

- 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

Patterns allow you to define a regular expression. A compiled regex pattern can speed up your program when the pattern is use frequently.

Matcher

- A matcher is a class in the java.util.regex package that stores a possible match between a Pattern and a String

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

- A Matcher is initialized as follows:

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

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

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

Pattern Matcher Example

1. Check that the String str conforms to the defined Pattern p, the matcher compares the str and pattern

```
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="AAAAAhhh";

      System.out.println(isMatch(str, p));
   }//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
```

Change the pattern to something of your choice and test using valid and invalid data.

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

RegEx Groups Example

1. Create the following class named RegExGroups

```
package regexgroups;

public class RegExGroups {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      String date;
      Pattern dateP = Pattern.compile("([0-9]{2})/([0-9]{2})/([0-9]{4})");

      in.close();
   }//end method main

   static String getDate(Scanner in, Pattern dateP) {
   }//end method getDate

}//end class RegExGroups
```

Group 1

Group 2

Group 3

Each group must be separated with a slash

Group 1 and Group 2 are defined to consist of 2 digits each.
Group 3 (the year) is defined to consist of 4 digits.

RegEx Groups Example

2. Update the getDate method to use a matcher object

```
static String getDate(Scanner in, Pattern dateP) {
   String date;
   Matcher dateM;
   do {
      System.out.print("Enter a Date (dd/mm/yyyy): ");
      date = in.nextLine();
      dateM = dateP.matcher(date);
      if(dateM.matches()){
         String day = dateM.group(1);
         String month = dateM.group(2);
         String year = dateM.group(3);
         date = month + "/" + day + "/" + year;
      }//endif
   }while(!dateM.matches());
   return date;
}//end method getDate
```

Group 1

Group 2

Group 3

It is still possible to get the whole Matcher by calling **`.group(0)`**.

RegEx Groups Example

3. Update the main method to call the getDate() method and display the updated date value in US style

```
public class RegExGroups {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      String date;
      Pattern dateP = Pattern.compile("([0-9]{2})/([0-9]{2})/([0-9]{4})");

      date = getDate(in, dateP);
      System.out.println("US style date - " + date);

      in.close();
   }//end method main
```

Test your code using a variety of dates both valid and invalid.

RegEx Groups Example

- Although the regular expression takes care of the format it does not check that the values are valid
- To stop accepting values out with the range of days and months, validation code will have to be added
- This can be done manually or through the use of the methods available in the DateFormat class
  - A simpleDate pattern can be provided that defines the order of the date elements
  - The String date can then be parsed
  - A try catch is used to define the Boolean return value

RegEx Groups Example

4. Create this method at the bottom of the class

```
   static boolean validateDate(String newDate) {
      DateFormat format = new SimpleDateFormat("dd/MM/yyyy");
      //Date must match the SimpleDate pattern exactly
      format.setLenient(false);
      String date = newDate;
      try {
         format.parse(date);
         return true;
      } catch (ParseException e) {
           System.out.println(date + " is not valid according to "
                  + ((SimpleDateFormat) format).toPattern() + " pattern.");
           return false;
        }//end try/catch
   }//end method validateDate
}//end class RegExGroups
```

RegEx Groups Example

5. Update the code in the do-while loop of the getDate method so that only valid dates are accepted

```
boolean validDate = false;
do {
   System.out.print("Enter a Date (dd/mm/yyyy): ");
   date = in.nextLine();
   dateM = dateP.matcher(date);
   if(dateM.matches()){
      String day = dateM.group(1);
      String month = dateM.group(2);
      String year = dateM.group(3);
      validDate = validateDate(date);
      if(dateM.matches() && validDate)
         date = month + "/" + day + "/" + year;
      else
         System.out.println("Incorrect date entered");
      }//endif
}while(!(dateM.matches() && validDate));
return date;
```

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

```
Returns true : Passw0rd
Returns false: Password
```

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, to split the String at any sequence of one or more digits, the code we could write something like this:

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

Combining String methods and regular expressions gives us many options for manipulating Strings.

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

Replacing with Regular Expressions

- Other methods that could be used are replaceFirst() and split() that can be both be researched through the Java API
  - String **replaceFirst**(String regex, String replacement) Replaces the first substring of this string that matches the given regular expression with the given replacement.
  - String[] **split**(String regex) Splits this string around matches of the given regular expression.
  - String[] **split**(String regex, int limit) Splits this string around matches of the given regular expression.

Replacing with Regular Expressions Example

1. 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

```
public class RegExReplace {
   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 RegExReplace
```

Output

```
help me I have no idea what's going on! ! !
```

This is very useful for dealing with user input as it removes unintended spaces.

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

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 pattern occurrences 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

ORACLE Academy


ORACLE
Academy