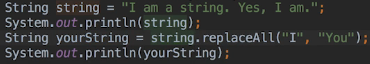
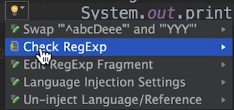
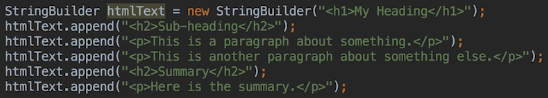
**Regular Expressions Introduction**  
\* Regular Expressions are a way to describe a string or a pattern.  
\* You’ve probably used Regular Expressions without knowing it. Some methods in the String class accept a Regular Expression as a parameter.  
\* For example: matches(), replaceAll(), split() method all actually work with Regular Expressions.  
\* RE are often used to search strings for a specific pattern or to validate that user input matches a specific pattern. For example checking user’s email input against a RE that describes how an email address string should look.  
\* **The simplest form of a RE is a** **String** **Literal**.  
\* **For example “Hello” is a RE**.  
\* **String** **replaceAll()** => accepts a RE that describes the pattern we want to replace as the 1st argument. The 2nd parameter is the actual replacement string.  
   
\* Now if all we could do was match String Literals in this way, RE wouldn’t really be very interesting. But we can do much more by using what’s called Character Classes and Boundary Matches.  
\* **Character** **Class** => like a wild card, it represents a set or class of characters.  
\* **Boundary Matcher** => looks for boundaries such as the beginning and end of a string or a word.  
\*  **.**  **Dot Character Class** matches any character.  
  
  
\* **^** **Caret Boundary Matcher** => always followed by a pattern which could be a String Literal or something more complex. When we use the ^, the RE must **match the beginning of the string**.  
  
  
\* Let’s now point out what I think is a really nifty feature of IntelliJ - when we place the cursor inside a RE, we see a light bulb popup on the left near the gutter, if we click on that:  
  
  
=> This is a really handy way to test a RE without actually having to run your application.  
\* However, it requires that the string as a whole must match the RE. So it doesn’t actually match parts of a string.  
\* **String** **matches()** => accepts a RE as a parameter and returns true if the string matches and false otherwise. The string as a whole must match the RE. This is the same behavior as the IntelliJ check.  
   
\* The documentation doesn’t really make it clear that the whole string has to match.  
\* It actually makes sense though - needing the full string to match - in the context of verifying user input when we want the entire input to match a given pattern.

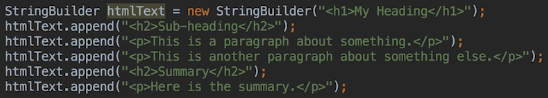
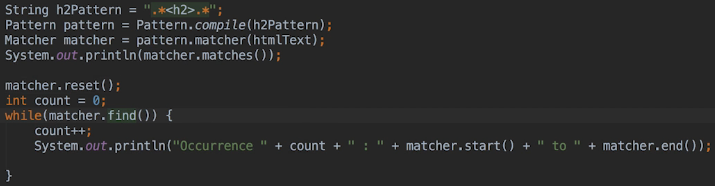
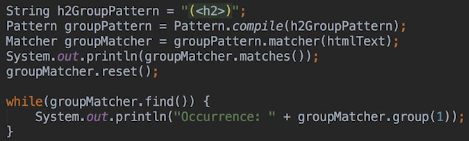
\* **$** => **Dollar Boundary Matcher** => always preceded by a pattern and it will match strings that end with the pattern. It’s the opposite of the ^ Boundary Matcher.  
  
  
\* When we want to match a specific letter or set of letters, we can put those letters within square brackets.  
\* **[]** => matches a specific letter or set of letters. Each individual character is examined.  
  
  
=> Matches every character that is `a` or `e` or `i`.  
\* We can use more [] together:  
  
  
=> Only gonna match if a/e/i is followed by F/j.

**Character Classes and Boundary Matchers**  
  
\* It’s saving us from having to upper or lower case the string first.  
\* If we want to replace everything ecept for e/j:  
  
  
\* **[^]** => When we use ^ inside [], it’s a Character Class, not a Boundary Matcher. It negates the pattern that follows it. So instead of matching all occurrences of e/j, it matches all that are not e/j.  
\* **[-]** => Range, matches all characters x-y, for example a-f or 3-8.  
  
  
\* It’s possible to turn off case sensitivity using a special construct.  
\* **(?i)** => Turns off the case sensitivity.  
  
  
\* This works with ASCII strings, if the string is Unicode, then we want to use:  
\* **(?iu)** => Turns off the case sensitivity for Unicode strings.  
\* **[0-9]** => Match all numbers.  
\* There’s a shorthand way to do the same thing.  
\* **\\d** => Match all numbers.  
  
\* I had to escape the \ because otherwise:  
  
\* **\\D** => Match all non-digits.  
  
\* **trim()** => remove whitespace from the beginning/end of a string.  
\* Let’s say we want to replace all whitespace everywhere in the string.  
\* **\\s** => match all whitespace.  
  


  
\* We can obviously use \t \n “ ” to replace specific whitespace:  
  
\* **\\S** => match all non-whitespace.  
\* **\\w** => match a-z A-Z 0-9 \_  
\* **\\W** => match anything except a-z A-Z 0-9 \_  
\* **\\b** => match **Word Boundaries**. It assumes that words are separates by whitespace.  
  
=> Each word has been surrounded by the replacement string.  
\* We could use it when we want to surround each word with tags of some kind. Perhaps you’re creating some HTML code or needing some sort of other tags.

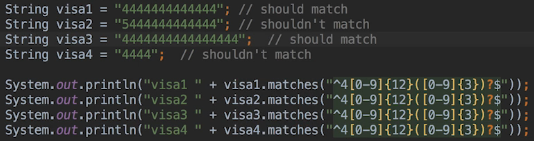
**Quantifiers and the Pattern and Matcher classes**  
\* There’s another way to write this expression:  
\* **Quantifier** => specifies how often an element in a RE can occur.  
\* **{7}** => indicate the number of the preceding character that must occur for it to be a match.  
  
\* The Quantifier always comes after the character that it applies to.  
\* Let’s suppose we don’t care how many `e`s there are, we want to match strings that begin with `abcD` followed by one or more `e`.  
\* **+** => Quantifier, match 1+ occurrence.  
  
\* Let’s say we don’t care about any `e`s. So we want to match if there are any `e`s and also no `e`.  
\* **\*** => Quantifier, match 0+ occurrence.  
  
\* One use for this might be when verifying user input and part of what we’ve asked for is optional.  
\* **{2,7}** => indicate the number range of the preceding character that must occur for it to be a match.  
  
\* Let’s try to replace all occurences of `h` followed by any number of `i` followed by 1 `j`.  
  
\* There are other Character Classes, Boundary Matchers and Quantifiers that we can use in RE.  
\* You can check the documentation for quite a complete list.  
\* **Some Java APIs want to work with Patterns rather than a string that represents a RE**.  
\* **Pattern.compile(“RE”)** => **compile a RE into a Pattern**.  
=> This is often done when we want to work with methods in the Matcher class.  
\* Matchers work with classes that implement the CharSequence interface and that means that we can use Matchers with String, StringBuffer, StringBuilder and other classes that implement that interface.  
\* Generally we use a **Matcher** **when we want to find multiple occurences of a pattern or when we want to use the same pattern with multiple sequences**.

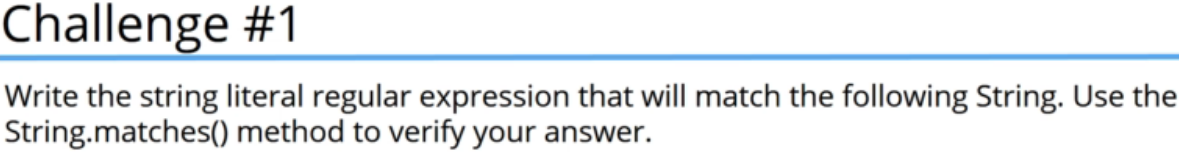
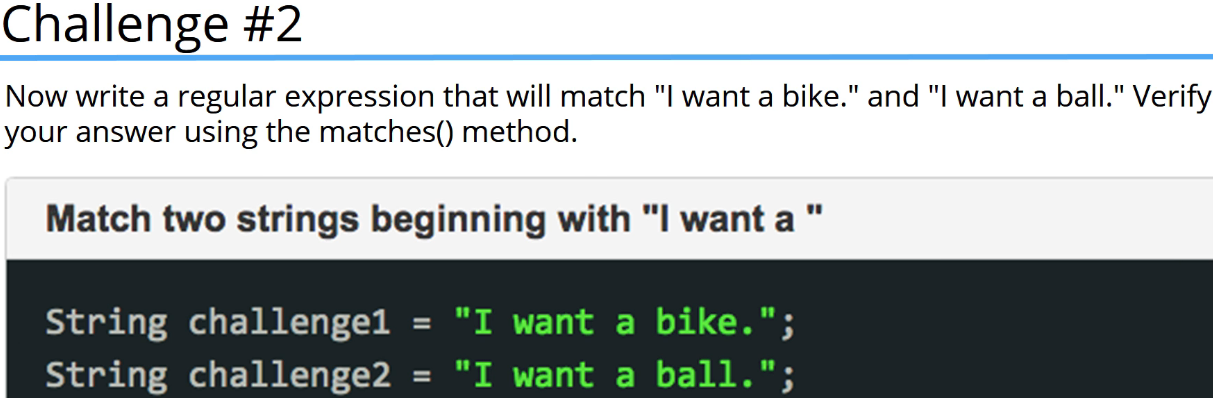
  
\* Let’s say we want to find all the occurrences of the <h2> tag in a CharSequence of HTML text.  
\* I’m using a StringBuilder because we’re creating a long string.  
\* **Matcher** **matcher()**\* **Pattern.CASE\_INSENSITIVE**  
\* **Pattern.UNICODE\_CASE** => to search Unicode strings  
  
\* We’re using the RE <h2> to find occurences of this tag. Next we compile() the Pattern. That creates a Patter instance thata we can use to create a Matcher instance. Finally we call the matches() method inside the Matcher’s class to check whether the Pattern matches the RE.  
\* Once you have a pattern, you can use that to create as many matchers as you want.   
\* We got false because it turns out that the matches() method wants to match the string as a whole. Just like the String matches() method.  
\* What we need to do is write a RE that’s gonna match the entire text.  
  
  
\* We’ll talk about finding how many occurences there are and where they occur in the next video.

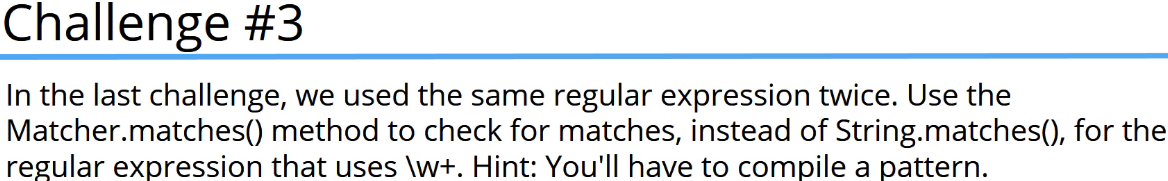
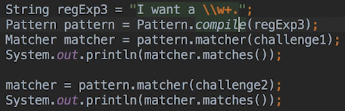
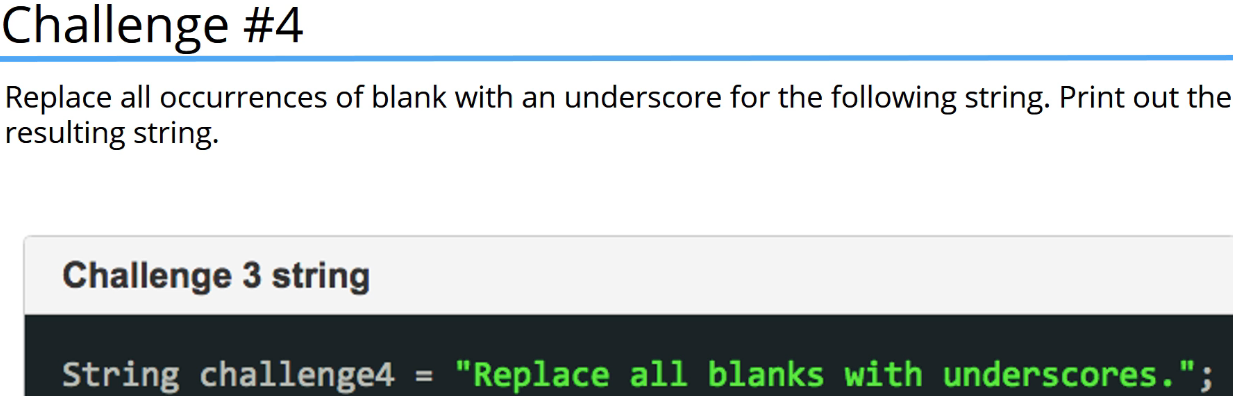
**Matcher find and Group Methods**  
\* **Matcher** **find()** => **looks for occurences of a pattern used to create the Matcher**.   
\* When it finds one, the internal state of the Matcher instance is updated and we can then call the:  
\* **Matcher** **start()** => returns the index of the first character in the match.  
\* **Matcher** **end()** => returns the index of the character that occurs right after the last character in the match.  
  
  
\* This won’t print anything, the matcher.find() will be false.  
=> That’s because we can only use a Matcher once. Matchers have an internal state, it’s updated whenever we use it. So the Matcher’s updated and then we can use the start() and then() methods. Buit when we call the matches() method so that we could print true/false to the console, the internal state of the matcher was updated in such a way that we have to reset the matcher before we can use it again.  
\* **Matcher** **reset()** => resets the internal state of a matcher.  
  
  
  
\* We used a broad expression with the matches() method because the entire text had to match.  
  
  
\* Keep in mind that the index returned by the end() method is the index for the first character that comes after a match.  
\* This obviously works but when you’re using a matcher to find multiple occurences of a pattern, there’s an easy way to do it.  
\* **()** => **Group**, we access them using the **group()** method in the Matcher class. One version of the method takes an int parameter which is the group number. When working with groups, **the entire Character Sequence is Group 0. So when we want to access a specific group, we start counting from the number 1**.  
  
  
  
  
\* It’s starting to look like what we want but not exactly what we want - the RE grabbed everything from the the first <h2> to the </h2> of the second one.  
\* The reason this is happening is due to a concept that I haven’t discussed yet.  
=> **Greedy** **Quantifiers** VS **Reluctant/Lazy Quantifiers**.  
\* **\*** => **Greedy Quantifier** => **it’ll grab as much as it can**.  
\* **?** => **Lazy Quantifier** => 0-1 occurences, **we can use it to turn a Greedy Quantifier into a Lazy one**.  
  
  
\* If we weren’t interested in empty h2 tags, we’d use the RE .+ instead of .\* to do that.  
\* We can also use multiple groups. If we just wanted the text between the h2, we could change it up a little bit:  
  
\* We now have 3 groups defined there.

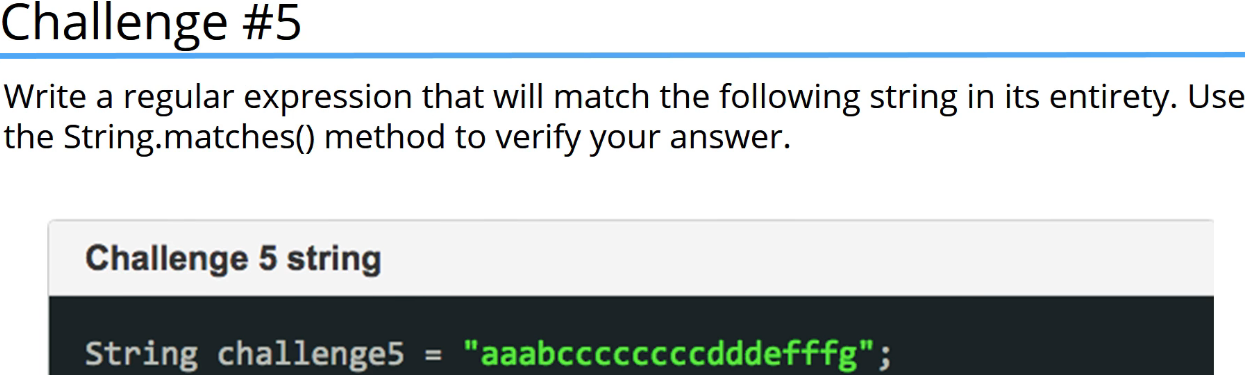
  
\* You can also use Matchers to find and replace parts of a string using the   
**replaceFirst()   
replaceAll()**   
methods.  
\* You can refer to the Matcher documentation for more information about other methods in the Matcher class and what you can do with them.

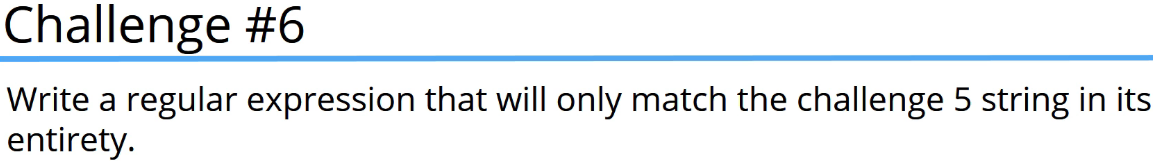
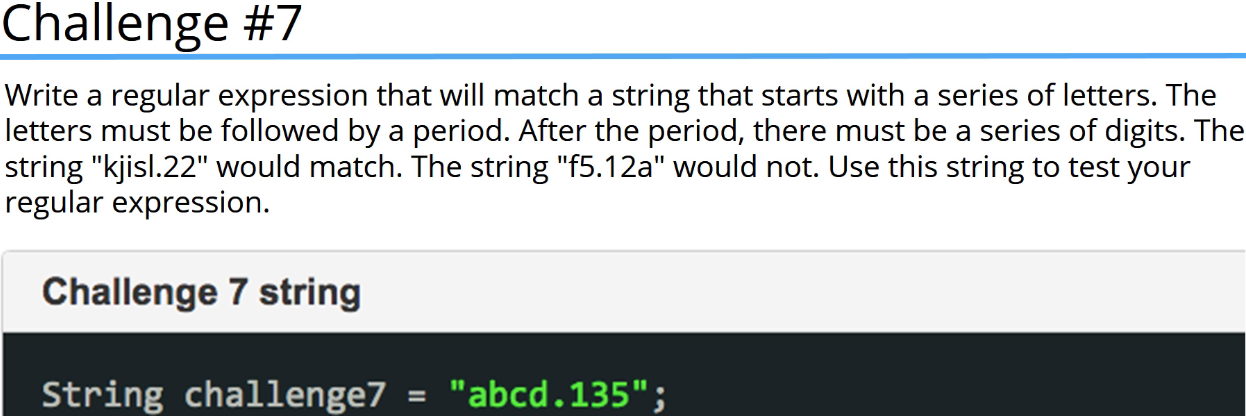
**And, Or & Not**  
\* We’ve already used the AND operator because it’s implicit.  
\* **and** => implicit  
 =   
\* **|** => or  
  
  
\* Not has 2 versions:  
\* **^** => not in []  
  
\* Let’s say we want to match occurences of T that aren’t followed by the character V.a  
\* First attempt:  
  
  
\* The single T at the end wasn’t found because when we use the **[^v]**, we must match a character in order for a RE to match. So we’re saying that the T **must be followed by any character other than V**.  
\* The [^v] consumes a character. It requires a character.  
\* **!** => not, has to be used in a Negative Look Ahead Expression. It can match nothing.  
\* The ? is part of the Look Ahead syntax.  
\* **(?!)** => **Negative Look Ahead Expression**.  
  
  
\* When we use a Look Ahead, the characters in the Look Ahead aren’t part of the match.  
\* Also the indices indicate that our matches are 1 character length. They’re no longer matchin T followed by another character anymore. They only match the T.  
=> **Look Ahead doesn’t consume the character that it matches. They don’t include the characters they match in the matched text**.  
\* **(?=)** => **Positive Look Ahead Expressions**.  
  
\* Find every T that is followed by V but don’t include the V in the result.

\* Let’s look at a couple of real world examples of using RE:  
=> **Validate user input**  
> US Phone Number, such as: (800) 123-4567  
  
  
  
\* There are many ways to write a RE that matches a phone number. For example we might not expect () around the area code. In a real world application we’d probably remove punctuation and whitespace from the phone number before processing it. And we might require the country code.  
=> **Checking for Visa Card**  
\* Visa Card numbers start with 4, new accounts have 16 numbers, older accounts have 13 numbers.  
  


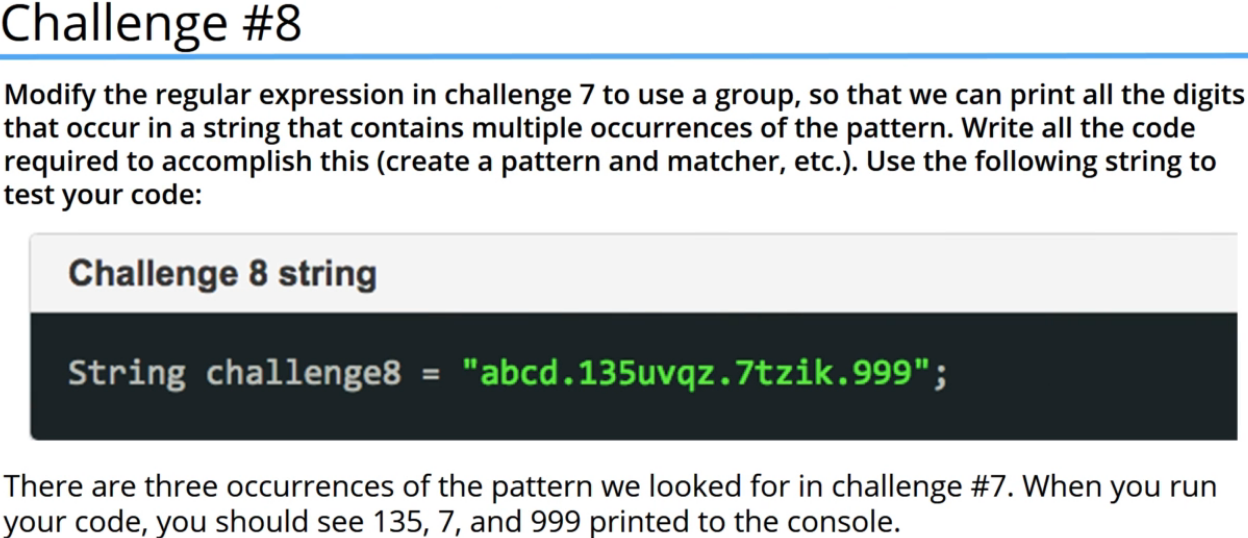
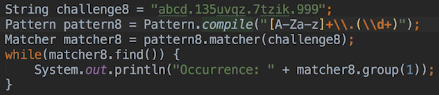
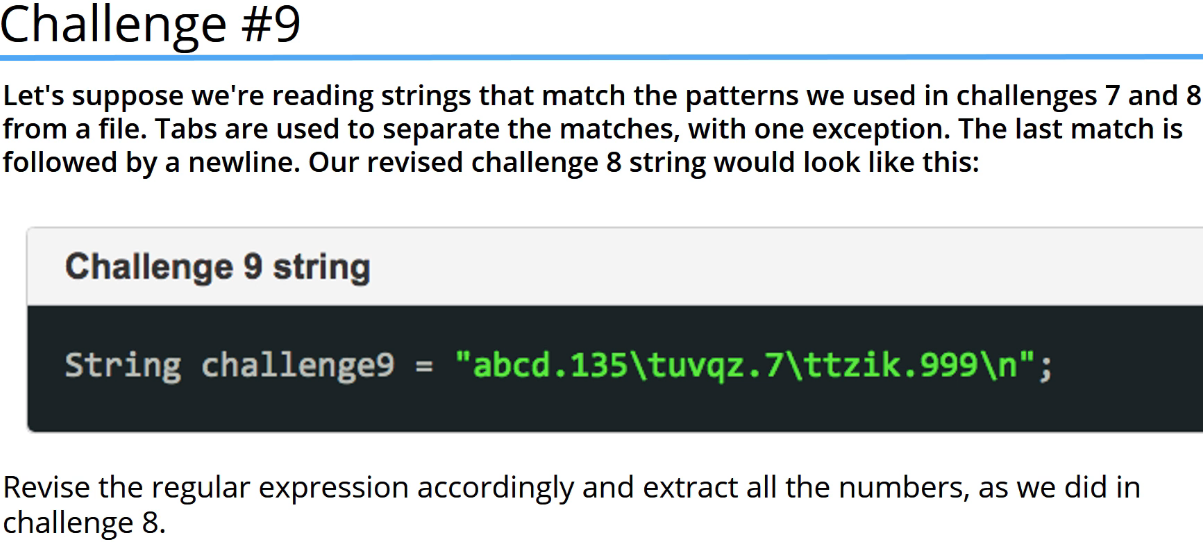
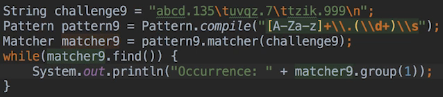
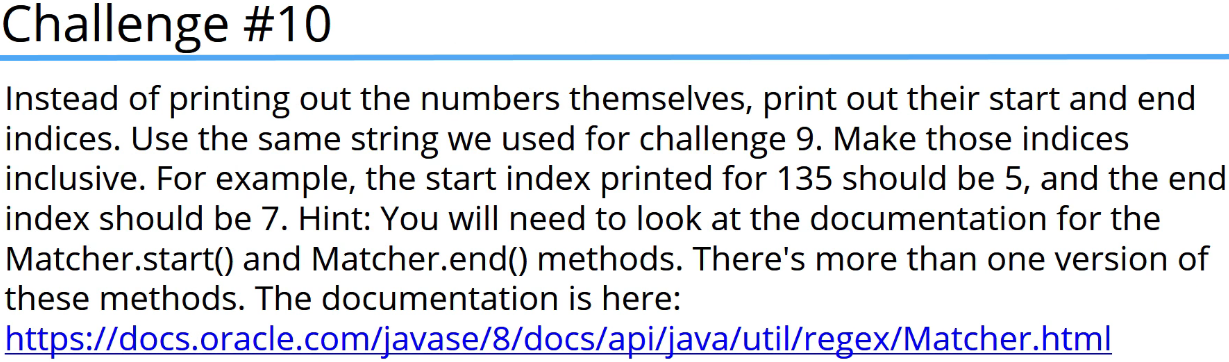
**Regular Expressions Challenge Part 1**  
\* There’s often more than one way to write a RE so my answers may not always be the only answer.  
\* You should verify your answers using   
String **matches()**   
String **replaceAll()**   
Matcher **matches()**  
  
  
 **OR**

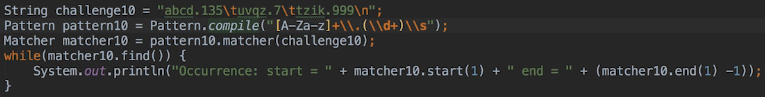
  
  
\* We can get a new matcher instance using the same pattern object in our challenge2 string. As a result we don’t have to reset the matcher becauase we’re using different matcher instances with each string.  


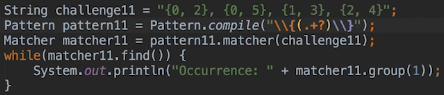
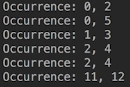
  


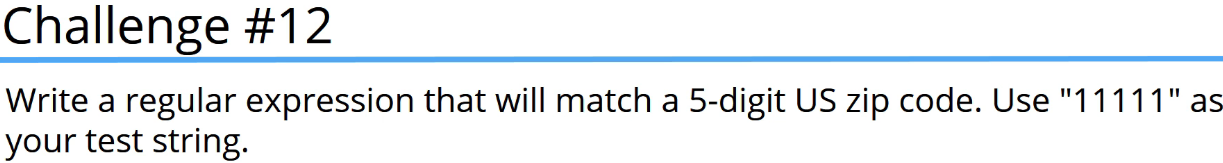
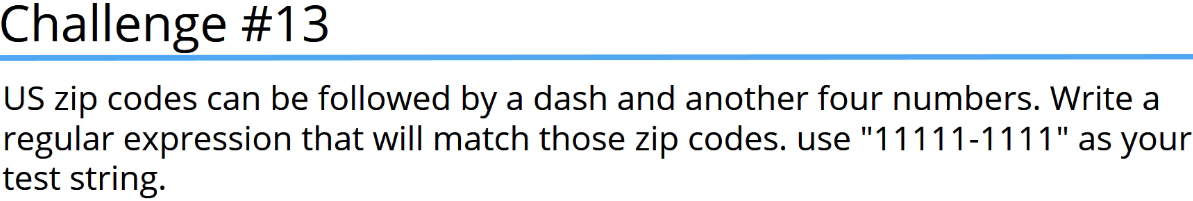
  
  
  
\* We can also verify using the replaceAll():  
  


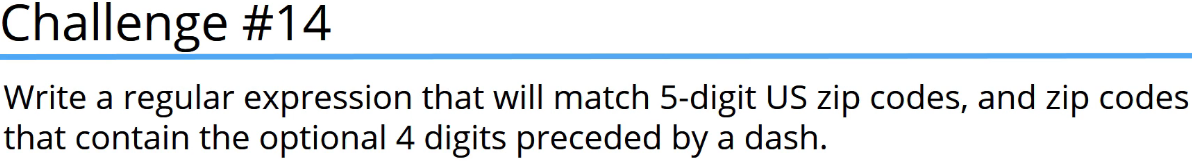
  
\* We could’ve used the Case Insensitivity modifier to only specify one range.

**Regular Expressions Challenge Part 2**  
  
  
  
  
\* The \\s is gonna catch the white space character that occurs after each occurrence of letters.digits.  



  
\* **?** => **Turns the + Quantifier into a Lazy Quantifier**. Prevents it from matching more characters than we want essentially. Will limit the match to the contents of {}.  
  
\* If we wanted to only match digits inside the {} separeted by a comma and a space:  
  
  
\* If we wanted to extract only the digits without comma and space, we’d put each set of digits into its own group.

**Regular Expressions Challenge Part 3**  
  
  
  
  
\* While verifying user input for zip codes, we could check the length of the input and from there decide which regular expression to use. But ideally it would be nice to have 1 RE that would match both cases.

  
  
\* I’ve modified the RE by enclosing the optional part within a group.  
=> The ()? means there can be 0 or 1 occurrences of the group.  
  
  
\* Whenever you would need to write a RE that matches a piece of data like a Zip Code, telephone number, email addres, etc., we can usually find REs that solve these problems on the web.  
\* But my suggestion here is to be careful because there’s a lot of variations and in some cases much debate about which one is correct.  
\* Also the REs can be wrong or incomplete.  
\* For example a RE for a Canadian Zip Code:  
  
=> Here it’s wrong because Canadia Post Codes can’t contain certain letters and they’re generally not written using a dash, although that’s not strictly wrong.  
\* The point here is to remember that not everything you see on the Internet is true. Especially with the Regular Expressions.

**Resources**  
Class Pattern  
<https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html>   
Class Matcher  
<https://docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html>