Using the Test Method

Regular expressions are used in programming languages to match parts of strings. You create patterns to help you do that matching.

If you want to find the word `the` in the string `The dog chased the cat`, you could use the following regular expression: `/the/`. Notice that quote marks are not required within the regular expression.

JavaScript has multiple ways to use regexes. One way to test a regex is using the `.test()` method. The `.test()` method takes the regex, applies it to a string (which is placed inside the parentheses), and returns `true` or `false` if your pattern finds something or not.

```
let testStr = "freeCodeCamp";
let testRegex = /Code/;
testRegex.test(testStr);
```

The `test` method here returns `true`.

Match Literal Strings

In the last challenge, you searched for the word `Hello` using the regular expression `/Hello/`. That regex searched for a literal match of the string `Hello`. Here's another example searching for a literal match of the string `Kevin`:

```
let testStr = "Hello, my name is Kevin.";
let testRegex = /Kevin/;
testRegex.test(testStr);
```

This `test` call will return `true`.

Any other forms of `Kevin` will not match. For example, the regex `/Kevin/` will not match `kevin` or `KEVIN`.

```
let wrongRegex = /kevin/;
wrongRegex.test(testStr);
```

This `test` call will return `false`.

A future challenge will show how to match those other forms as well.

Ignore Case While Matching

Up until now, you've looked at regexes to do literal matches of strings. But sometimes, you might want to also match case differences.

Case (or sometimes letter case) is the difference between uppercase letters and lowercase letters. Examples of uppercase are `A`, `B`, and `C`. Examples of lowercase are `a`, `b`, and `c`.

You can match both cases using what is called a flag. There are other flags but here you'll focus on the flag that ignores case - the `i` flag. You can use it by appending it to the regex. An example of using this flag is `/ignorecase/i`. This regex can match the strings `ignorecase`, `igNoreCase`, and `IgnoreCase`.

Match a Literal String with Different Possibilities

Using regexes like `/coding/`, you can look for the pattern `coding` in another string.

This is powerful to search single strings, but it's limited to only one pattern. You can search for multiple patterns using the `alternation` or `OR` operator: `|`.

This operator matches patterns either before or after it. For example, if you wanted to match the strings `yes` or `no`, the regex you want is `/yes|no/`.

You can also search for more than just two patterns. You can do this by adding more patterns with more `OR` operators separating them, like `/yes|no|maybe/`.

Extract Matches

So far, you have only been checking if a pattern exists or not within a string. You can also extract the actual matches you found with the `.match()` method.

To use the `.match()` method, apply the method on a string and pass in the regex inside the parentheses.

Here's an example:

```
"Hello, World!".match(/Hello/);
let ourStr = "Regular expressions";
let ourRegex = /expressions/;
ourStr.match(ourRegex);
```

Here the first `match` would return `["Hello"]` and the second would return `["expressions"]`.

Note that the `.match` syntax is the "opposite" of the `.test` method you have been using thus far:

```
'string'.match(/regex/);
/regex/.test('string');
```

Find More Than the First Match

So far, you have only been able to extract or search a pattern once.

```
let testStr = "Repeat, Repeat, Repeat";
let ourRegex = /Repeat/;
testStr.match(ourRegex);
```

Here `match` would return `["Repeat"]`.

To search or extract a pattern more than once, you can use the `g` flag.

```
let repeatRegex = /Repeat/g;
testStr.match(repeatRegex);
```

And here `match` returns the value `["Repeat", "Repeat", "Repeat"]`

Match Anything with Wildcard Period

Sometimes you won't (or don't need to) know the exact characters in your patterns. Thinking of all words that match, say, a misspelling would take a long time. Luckily, you can save time using the wildcard character: `.`

The wildcard character `.` will match any one character. The wildcard is also called `dot` and `period`. You can use the wildcard character just like any other character in the regex. For example, if you wanted to match `hug`, `huh`, `hut`, and `hum`, you can use the regex `/hu./` to match all four words.

```
let humStr = "I'll hum a song";
let hugStr = "Bear hug";
let huRegex = /hu./;
huRegex.test(humStr);
huRegex.test(hugStr);
```

Both of these `test` calls would return `true`.

Match Single Character with Multiple Possibilities

You learned how to match literal patterns (`/literal/`) and wildcard character (`/./`). Those are the extremes of regular expressions, where one finds exact matches and the other matches everything. There are options that are a balance between the two extremes.

You can search for a literal pattern with some flexibility with *character classes*. Character classes allow you to define a group of characters you wish to match by placing them inside square (`[` and `]`) brackets.

For example, you want to match `bag`, `big`, and `bug` but not `bog`. You can create the regex `/b[aiu]g/` to do this. The `[aiu]` is the character class that will only match the characters `a`, `i`, or `u`.

```
let bigStr = "big";
let bagStr = "bag";
let bugStr = "bug";
let bogStr = "bog";
let bgRegex = /b[aiu]g/;
bigStr.match(bgRegex);
bagStr.match(bgRegex);
bugStr.match(bgRegex);
bogStr.match(bgRegex);
```

In order, the four `match` calls would return the values `["big"]`, `["bag"]`, `["bug"]`, and `null`.

Match Letters of the Alphabet

You saw how you can use *character sets* to specify a group of characters to match, but that's a lot of typing when you need to match a large range of characters (for example, every letter in the alphabet). Fortunately, there is a built-in feature that makes this short and simple.

Inside a character set, you can define a range of characters to match using a hyphen character: `-`.

For example, to match lowercase letters `a` through `e` you would use `[a-e]`.

```
let catStr = "cat";
let batStr = "bat";
let matStr = "mat";
let bgRegex = /[a-e]at/;
```

```
catStr.match(bgRegex);
batStr.match(bgRegex);
matStr.match(bgRegex);
```

In order, the three `match` calls would return the values `["cat"]`, `["bat"]`, and `null`.

Match Numbers and Letters of the Alphabet

Using the hyphen (`-`) to match a range of characters is not limited to letters. It also works to match a range of numbers.

For example, `/[0-5]/` matches any number between `0` and `5`, including the `0` and `5`.

Also, it is possible to combine a range of letters and numbers in a single character set.

```
let jennyStr = "Jenny8675309";
let myRegex = /[a-z0-9]/ig;
jennyStr.match(myRegex);
```

Match Single Characters Not Specified

So far, you have created a set of characters that you want to match, but you could also create a set of characters that you do not want to match. These types of character sets are called *negated character sets*.

To create a negated character set, you place a caret character (`^`) after the opening bracket and before the characters you do not want to match.

For example, `/[^aeiou]/gi` matches all characters that are not a vowel. Note that characters like `.`, `!`, `[`, `@`, `/` and white space are matched - the negated vowel character set only excludes the vowel characters.

Match Characters that Occur One or More Times

Sometimes, you need to match a character (or group of characters) that appears one or more times in a row. This means it occurs at least once, and may be repeated.

You can use the `+` character to check if that is the case. Remember, the character or pattern has to be present consecutively. That is, the character has to repeat one after the other.

For example, `/a+/g` would find one match in `abc` and return `["a"]`. Because of the `+`, it would also find a single match in `aabc` and return `["aa"]`.

If it were instead checking the string `abab`, it would find two matches and return `["a", "a"]` because the `a` characters are not in a row - there is a `b` between them. Finally, since there is no `a` in the string `bcd`, it wouldn't find a match.

Match Characters that Occur Zero or More Times

The last challenge used the plus `+` sign to look for characters that occur one or more times. There's also an option that matches characters that occur zero or more times.

The character to do this is the asterisk or star: `*`.

```
let soccerWord = "gooooooooal!";
let gPhrase = "gut feeling";
let oPhrase = "over the moon";
let goRegex = /go*/;
soccerWord.match(goRegex);
gPhrase.match(goRegex);
oPhrase.match(goRegex);
```

In order, the three `match` calls would return the values `["goooooooo"]`, `["g"]`, and `null`.

Find Characters with Lazy Matching

In regular expressions, a *greedy* match finds the longest possible part of a string that fits the regex pattern and returns it as a match. The alternative is called a *lazy* match, which finds the smallest possible part of the string that satisfies the regex pattern.

You can apply the regex `/t[a-z]*i/` to the string `"titanic"`. This regex is basically a pattern that starts with `t`, ends with `i`, and has some letters in between.

Regular expressions are by default greedy, so the match would return `["titani"]`. It finds the largest sub-string possible to fit the pattern.

However, you can use the `?` character to change it to lazy matching. `"titanic"` matched against the adjusted regex of `/t[a-z]*?i/` returns `["ti"]`.

Note: Parsing HTML with regular expressions should be avoided, but pattern matching an HTML string with regular expressions is completely fine.

Match Beginning String Patterns

Prior challenges showed that regular expressions can be used to look for a number of matches. They are also used to search for patterns in specific positions in strings.

In an earlier challenge, you used the caret character (`^`) inside a character set to create a negated character set in the form `[^thingsThatWillNotBeMatched]`. Outside of a character set, the caret is used to search for patterns at the beginning of strings.

```
let firstString = "Ricky is first and can be found.";
let firstRegex = /^Ricky/;
firstRegex.test(firstString);
let notFirst = "You can't find Ricky now.";
firstRegex.test(notFirst);
```

The first `test` call would return `true`, while the second would return `false`.

Match Ending String Patterns

In the last challenge, you learned to use the caret character to search for patterns at the beginning of strings. There is also a way to search for patterns at the end of strings.

You can search the end of strings using the dollar sign character `$` at the end of the regex.

```
let theEnding = "This is a never ending story";
let storyRegex = /story$/;
storyRegex.test(theEnding);
let noEnding = "Sometimes a story will have to end";
storyRegex.test(noEnding);
```

The first `test` call would return `true`, while the second would return `false`.

Match All Letters and Numbers

Using character classes, you were able to search for all letters of the alphabet with `[a-z]`. This kind of character class is common enough that there is a shortcut for it, although it includes a few extra characters as well.

The closest character class in JavaScript to match the alphabet is `\w`. This shortcut is equal to `[A-Za-z0-9_]`. This character class matches upper and lowercase letters plus numbers. Note, this character class also includes the underscore character (_).

```
let longHand = /[A-Za-z0-9_]+/;
let shortHand = /\w+/;
let numbers = "42";
let varNames = "important_var";
longHand.test(numbers);
shortHand.test(numbers);
longHand.test(varNames);
shortHand.test(varNames);
```

All four of these `test` calls would return `true`.

These shortcut character classes are also known as *shorthand character classes*.

Match All Numbers

You've learned shortcuts for common string patterns like alphanumerics. Another common pattern is looking for just digits or numbers.

The shortcut to look for digit characters is `\d`, with a lowercase `d`. This is equal to the character class `[0-9]`, which looks for a single character of any number between zero and nine.

Match All Non-Numbers

The last challenge showed how to search for digits using the shortcut `\d` with a lowercase `d`. You can also search for non-digits using a similar shortcut that uses an uppercase `D` instead.

The shortcut to look for non-digit characters is `\D`. This is equal to the character class `[^0-9]`, which looks for a single character that is not a number between zero and nine.

Match Whitespace

The challenges so far have covered matching letters of the alphabet and numbers. You can also match the whitespace or spaces between letters.

You can search for whitespace using `\s`, which is a lowercase `s`. This pattern not only matches whitespace, but also carriage return, tab, form feed, and new line characters. You can think of it as similar to the character class `[ \r\t\f\n\v]`.

```
let whiteSpace = "Whitespace. Whitespace everywhere!"
let spaceRegex = /\s/g;
whiteSpace.match(spaceRegex);
```

This `match` call would return `[" ", " "]`.

Match Non-Whitespace Characters

You learned about searching for whitespace using `\s`, with a lowercase `s`. You can also search for everything except whitespace.

Search for non-whitespace using `\S`, which is an uppercase `s`. This pattern will not match whitespace, carriage return, tab, form feed, and new line characters. You can think of it being similar to the character class `[^ \r\t\f\n\v]`.

```
let whiteSpace = "Whitespace. Whitespace everywhere!"
let nonSpaceRegex = /\S/g;
whiteSpace.match(nonSpaceRegex).length;
```

The value returned by the `.length` method would be `32`.

Specify Upper and Lower Number of Matches

Recall that you use the plus sign `+` to look for one or more characters and the asterisk `*` to look for zero or more characters. These are convenient but sometimes you want to match a certain range of patterns.

You can specify the lower and upper number of patterns with *quantity specifiers*. Quantity specifiers are used with curly brackets (`{` and `}`). You put two numbers between the curly brackets - for the lower and upper number of patterns.

For example, to match only the letter `a` appearing between `3` and `5` times in the string `ah`, your regex would be `/a{3,5}h/`.

```
let A4 = "aaaah";
let A2 = "aah";
let multipleA = /a{3,5}h/;
multipleA.test(A4);
multipleA.test(A2);
```

The first `test` call would return `true`, while the second would return `false`.

Specify Only the Lower Number of Matches

You can specify the lower and upper number of patterns with quantity specifiers using curly brackets. Sometimes you only want to specify the lower number of patterns with no upper limit.

To only specify the lower number of patterns, keep the first number followed by a comma.

For example, to match only the string `hah` with the letter `a` appearing at least `3` times, your regex would be `/ha{3,}h/`.

```
let A4 = "haaaah";
let A2 = "haah";
let A100 = "h" + "a".repeat(100) + "h";
let multipleA = /ha{3,}h/;
multipleA.test(A4);
multipleA.test(A2);
multipleA.test(A100);
```

In order, the three `test` calls would return `true`, `false`, and `true`.

Specify Exact Number of Matches

You can specify the lower and upper number of patterns with quantity specifiers using curly brackets. Sometimes you only want a specific number of matches.

To specify a certain number of patterns, just have that one number between the curly brackets.

For example, to match only the word `hah` with the letter `a` `3` times, your regex would be `/ha{3}h/`.

```
let A4 = "haaaah";
let A3 = "haaah";
let A100 = "h" + "a".repeat(100) + "h";
let multipleHA = /ha{3}h/;
multipleHA.test(A4);
multipleHA.test(A3);
multipleHA.test(A100);
```

In order, the three `test` calls would return `false`, `true`, and `false`.

Check for All or None

Sometimes the patterns you want to search for may have parts of it that may or may not exist. However, it may be important to check for them nonetheless.

You can specify the possible existence of an element with a question mark, `?`. This checks for zero or one of the preceding element. You can think of this symbol as saying the previous element is optional.

For example, there are slight differences in American and British English and you can use the question mark to match both spellings.

```
let american = "color";
let british = "colour";
let rainbowRegex= /colou?r/;
rainbowRegex.test(american);
rainbowRegex.test(british);
```

Both uses of the `test` method would return `true`.

Positive and Negative Lookahead

Lookaheads are patterns that tell JavaScript to look-ahead in your string to check for patterns further along. This can be useful when you want to search for multiple patterns over the same string.

There are two kinds of lookaheads: *positive lookahead* and *negative lookahead*.

A positive lookahead will look to make sure the element in the search pattern is there, but won't actually match it. A positive lookahead is used as `(?=...)` where the `...` is the required part that is not matched.

On the other hand, a negative lookahead will look to make sure the element in the search pattern is not there. A negative lookahead is used as `(?!...)` where the `...` is the pattern that you do not want to be there. The rest of the pattern is returned if the negative lookahead part is not present.

Lookaheads are a bit confusing but some examples will help.

```
let quit = "qu";
let noquit = "qt";
let quRegex= /q(?=u)/;
let qRegex = /q(?!u)/;
quit.match(quRegex);
```

```
noquit.match(qRegex);
```

Both of these `match` calls would return `["q"]`.

A more practical use of lookaheads is to check two or more patterns in one string. Here is a (naively) simple password checker that looks for between 3 and 6 characters and at least one number:

```
let password = "abc123";
let checkPass = /(?=\w{3,6})(?=\D*\d)/;
checkPass.test(password);
```

Check For Mixed Grouping of Characters

Sometimes we want to check for groups of characters using a Regular Expression and to achieve that we use parentheses `()`.

If you want to find either `Penguin` or `Pumpkin` in a string, you can use the following Regular Expression: `/P(engu|umpk)in/g`

Then check whether the desired string groups are in the test string by using the `test()` method.

```
let testStr = "Pumpkin";
let testRegex = /P(engu|umpk)in/;
testRegex.test(testStr);
```

The `test` method here would return `true`.

Reuse Patterns Using Capture Groups

Some patterns you search for will occur multiple times in a string. It is wasteful to manually repeat that regex. There is a better way to specify when you have multiple repeat substrings in your string.

You can search for repeat substrings using *capture groups*. Parentheses, `(` and `)`, are used to find repeat substrings. You put the regex of the pattern that will repeat in between the parentheses.

To specify where that repeat string will appear, you use a backslash (`\`) and then a number. This number starts at 1 and increases with each additional capture group you use. An example would be `\1` to match the first group.

The example below matches any word that occurs twice separated by a space:

```
let repeatStr = "regex regex";
let repeatRegex = /(\w+)\s\1/;
repeatRegex.test(repeatStr);
repeatStr.match(repeatRegex);
```

The `test` call would return `true`, and the `match` call would return `["regex regex", "regex"]`.

Using the `.match()` method on a string will return an array with the string it matches, along with its capture group.

```
let repeatNum = "42 42 42";
let reRegex = /^(\d+)\s\1\s\1$/;
let result = reRegex.test(repeatNum);
```

Use Capture Groups to Search and Replace

Searching is useful. However, you can make searching even more powerful when it also changes (or replaces) the text you match.

You can search and replace text in a string using `.replace()` on a string. The inputs for `.replace()` is first the regex pattern you want to search for. The second parameter is the string to replace the match or a function to do something.

```
let wrongText = "The sky is silver.";
let silverRegex = /silver/;
wrongText.replace(silverRegex, "blue");
```

The `replace` call would return the string `The sky is blue.`.

You can also access capture groups in the replacement string with dollar signs (`$`).

```
"Code Camp".replace(/(\w+)\s(\w+)/, '$2 $1');
```

The `replace` call would return the string `Camp Code`.