## WHAT IS SOFTWARE ENGINEERING?

As a branch of computer science, software engineering includes the development and building of computer systems software and applications software. Systems software consists of programs that include computing utilities and operations systems and applications software includes user-focused programs, including database programs, Web browsers and more. Knowledge of programming languages, software development and computer operating systems are three major components of the field. Software engineering is a related, but slightly different occupational field than computer hardware engineering, which is concerned with the design and development of hardware and computing technologies (e.g. processor design, networking, etc.)

Some of the major branches and career paths in software engineering include:

APPLICATIONS DEVELOPMENT

Problem solving-based, non-Web-based software development that includes programming languages such as Java and C#.

SYSTEMS DEVELOPMENT

Designing and coding background software created to support application development; includes program languages like C and C++.

WEB DEVELOPMENT

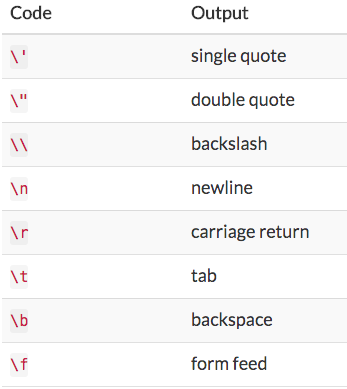
Designing software or applications to run in a Web browser and used programming languages such as HTML, JavaScript and PHP.

EMBEDDED SYSTEMS DEVELOPMENT

Designing computing systems and software to work on non-computing devices, such as automobiles. Uses programming languages such as C and assembly languages.

**FCC - Javascript**

In computer science, *data* is anything that is meaningful to the computer. JavaScript provides seven different *data types* which are undefined, null, boolean, string, symbol, number, and object.

* 

Strings:

* : So if var firstName = "Charles," you can get the value of the first letter of the string by using firstName[0]
* In JavaScript, String values are *immutable*, which means that they cannot be altered once created.
  + Note that this does *not* mean that myStr cannot be changed, just that the individual characters of a *string literal* cannot be changed. The only way to change myStr would be to assign it with a new string, like this

var myStr = "Bob";  
myStr = "Job";

* In order to get the last letter of a string, you can subtract one from the string's length.
* For example, if var firstName = "Charles", you can get the value of the last letter of the string by using firstName[firstName.length - 1]

Arrays

* You can also nest arrays within other arrays, like this: [["Bulls", 23], ["White Sox", 45]]. This is also called a Multi-dimensional Array.
* We can access data inside an array using indexes. Array indexes are written in the same bracket notation that strings use, except that instead of specifying a character, they are specifying an entry in the array.
* Create a variable called myData and set it to equal the first value of myArray using bracket notation.

var myData = myArray[0]

* One way to think of a multi-dimensional array, is as an array of arrays.

**Example**

var arr = [  
  [1,2,3],  
  [4,5,6],  
  [7,8,9],  
  [[10,11,12], 13, 14]  
];  
arr[3]; // equals [[10,11,12], 13, 14]  
arr[3][0]; // equals [10,11,12]  
arr[3][0][1]; // equals 11

* An easy way to append data to the end of an array is via the push()function.
* .push() takes one or more parameters and "pushes" them onto the end of the array.

var arr = [1,2,3];  
arr.push(4);  
// arr is now [1,2,3,4]

* Another way to change the data in an array is with the .pop()function.
* .pop()is used to "pop" a value off of the end of an array. We can store this "popped off" value by assigning it to a variable. In other words, .pop()removes the last element from an array and returns that element.

var threeArr = [1, 4, 6];  
var oneDown = threeArr.pop();  
console.log(oneDown); // Returns 6  
console.log(threeArr); // Returns [1, 4

* pop() always removes the last element of an array. What if you want to remove the first?
* That's where .shift()comes in. It works just like .pop(), except it removes the first element instead of the last.
* Not only can you shift elements off of the beginning of an array, you can also unshift elements to the beginning of an array i.e. add elements in front of the array.
* .unshift()works exactly like .push(), but instead of adding the element at the end of the array, unshift()adds the element at the beginning of the array.

Function

* Variables which are used without the var keyword are automatically created in the global scope. This can create unintended consequences elsewhere in your code or when running a function again. You should always declare your variables with var.
* It is possible to have both local and global variables with the same name. When you do this, the localvariable takes precedence over the globalvariable.
* In JavaScript, you can determine the type of a variable or a value with the typeof operator, as follows:
* typeof 3 // returns 'number'  
  typeof '3' // returns 'string'

Switch Statements and if/else

* Since === returns true or false, we can return the result of the comparison:

function isEqual(a,b) {

return a === b;

}

## Basic JavaScript: Build JavaScript Objects

Objects are similar to arrays, except that instead of using indexes to access and modify their data, you access the data in objects through what are called properties. Objects are useful for storing data in a structured way, and can represent real world objects, like a cat.

Heres a sample cat object:

var cat = {  
  "name": "Whiskers",  
  "legs": 4,  
  "tails": 1,  
  "enemies": ["Water", "Dogs"]  
};

There are two ways to access the properties of an object: dot notation (.) and bracket notation ([]), similar to an array.

Objects can be thought of as a key/value storage, like a dictionary. If you have tabular data, you can use an object to "lookup" values rather than a switch statement or an if/else chain. This is most useful when you know that your input data is limited to a certain range.

## Basic JavaScript: Manipulating Complex Objects

Sometimes you may want to store data in a flexible Data Structure. A JavaScript object is one way to handle flexible data. They allow for arbitrary combinations of strings, numbers, booleans, arrays, functions, and objects.

Here's an example of a complex data structure:

var ourMusic = [  
  {  
    "artist": "Daft Punk",  
    "title": "Homework",  
    "release\_year": 1997,  
    "formats": [   
      "CD",   
      "Cassette",   
      "LP"  
    ],  
    "gold": true  
  }  
];

This is an array which contains one object inside. The object has various pieces of metadata about an album. It also has a nested "formats"array. If you want to add more album records, you can do this by adding records to the top level array.

Objects hold data in a property, which has a key-value format. In the example above, "artist": "Daft Punk"is a property that has a key of "artist" and a value of "Daft Punk".

[JavaScript Object Notation](http://www.json.org/) or JSONis a related data interchange format used to store data.

## Basic JavaScript: Iterate with JavaScript While Loops

The first type of loop we will learn is called a "while" loop because it runs "while" a specified condition is true and stops once that condition is no longer true.

var ourArray = [];  
var i = 0;  
while(i < 5) {  
  ourArray.push(i);  
  i++;  
}

FOR LOOPS

You can run the same code multiple times by using a loop. The most common type of JavaScript loop is called a "for loop" because it runs "for" a specific number of times. For loops are declared with three optional expressions separated by semicolons:

for ([initialization]; [condition]; [final-expression])

## Iterate Through an Array with a For Loop

A common task in JavaScript is to iterate through the contents of an array. One way to do that is with a for loop. This code will output each element of the array arr to the console:

var arr = [10,9,8,7,6];  
for (var i = 0; i < arr.length; i++) {  
   console.log(arr[i]);  
}

Remember that Arrays have zero-based numbering, which means the last index of the array is length - 1. Our condition for this loop is i < arr.length, which stops when I is at length - 1.

## Basic JavaScript: Nesting For Loops

If you have a multi-dimensional array, you can use the same logic as the prior waypoint to loop through both the array and any sub-arrays. Here is an example:

var arr = [  
  [1,2], [3,4], [5,6]  
];  
for (var i=0; i < arr.length; i++) {  
  for (var j=0; j < arr[i].length; j++) {  
    console.log(arr[i][j]);  
  }  
}

This outputs each sub-element in arr one at a time. Note that for the inner loop, we are checking the .length of arr[i], since arr[i] is itself an array.

## Iterate with JavaScript Do...While Loops

You can run the same code multiple times by using a loop.

The next type of loop you will learn is called a "do...while" loop because it first will "do" one pass of the code inside the loop no matter what, and then it runs "while" a specified condition is true and stops once that condition is no longer true. Let's look at an example.

var ourArray = [];  
var i = 0;  
do {  
  ourArray.push(i);  
  i++;  
} while (i < 5);

This behaves just as you would expect with any other type of loop, and the resulting array will look like [0, 1, 2, 3, 4]. However, what makes the do...while different from other loops is how it behaves when the condition fails on the first check. Let's see this in action.

Essentially, a do...while loop ensures that the code inside the loop will run at least once.

Let's try getting a do...while loop to work by pushing values to an array.

Change the while loop in the code to a do...while loop so that the loop will push the number 10 to myArray, and I will be equal to 11 when your code finishes running.

## Generate Random Fractions with JavaScript

Random numbers are useful for creating random behavior.

JavaScript has a Math.random() function that generates a random decimal number between 0(inclusive) and not quite up to 1(exclusive). Thus Math.random()can return a 0 but never quite return a 1

Use another function, Math.floor()to round the number down to its nearest whole number.

Math.floor(Math.random() \* 20); //generates a whole number from 0-19

Instead of generating a random number between zero and a given number like we did before, we can generate a random number that falls within a range of two specific numbers.

To do this, we'll define a minimum number min and a maximum number max.

## Basic JavaScript: Use the parseInt Function

The parseInt() function parses a string and returns an integer. Here's an example:

var a = parseInt("007");

WITH A RADIX

If the first character in the string can't be converted into a number, then it returns NaN

The parseInt() function parses a string and returns an integer. It takes a second argument for the radix, which specifies the base of the number in the string. The radix can be an integer between 2 and 36.

The function call looks like:

parseInt(string, radix);

And here's an example:

var a = parseInt("11", 2);

The radix variable says that "11" is in the binary system, or base 2. This example converts the string "11" to an integer 3.

## Use the Conditional (Ternary) Operator

The conditional operator, also called the ternary operator, can be used as a one line if-else expression.

The syntax is:

condition ? statement-if-true : statement-if-false;

MULTIPLE

function findGreaterOrEqual(a, b) {  
  if(a === b) {  
    return "a and b are equal";  
  }  
  else if(a > b) {  
    return "a is greater";  
  }  
  else {  
    return "b is greater";  
  }  
}

The above function can be re-written using multiple conditional operators:

function findGreaterOrEqual(a, b) {  
  return (a === b) ? "a and b are equal" : (a > b) ? "a is greater" : "b is greater";  
}

## ES6: Explore Differences Between the var and let Keywords

One of the biggest problems with declaring variables with the var keyword is that you can overwrite variable declarations without an error. In a small application, you might not run into this type of problem, but when your code becomes larger, you might accidentally overwrite a variable that you did not intend to overwrite.

Because this behavior does not throw an error, searching and fixing bugs becomes more difficult

A new keyword called let was introduced in ES6 to solve this potential issue with the var keyword. So unlike var, when using let, a variable with the same name can only be declared once.

Note the "use strict". This enables Strict Mode, which catches common coding mistakes and "unsafe" actions. For instance:

## Compare Scopes of the var and let Keywords

When you declare a variable with the var keyword, it is declared globally, or locally if declared inside a function.

The let keyword behaves similarly, but with some extra features. When you declare a variable with the let keyword inside a block, statement, or expression, its scope is limited to that block, statement, or expression.

**By using let you can declare variables in relation to their scope.**

function checkScope() {

"use strict";

let i = "function scope";

if (true) {

let i = "block scope";

console.log("Block scope i is: ", i);

}

console.log("Function scope i is: ", i);

return i;

}

ES6: Declare a Read-Only Variable with the const Keyword

Const has all the awesome features that let has, with the added bonus that variables declared using const are read-only. They are a constant value, which means that once a variable is assigned with const, it cannot be reassigned.

A common practice when naming constants is to use all uppercase letters, with words separated by an underscore. Const SENTENCE = “Gideon is awesome”

The const declaration has many use cases in modern JavaScript.

Some developers prefer to assign all their variables using const by default, unless they know they will need to reassign the value. Only in that case, they use let.

## ES6: Prevent Object Mutation

As seen in the previous challenge, const declaration alone doesn't really protect your data from mutation. To ensure your data doesn't change, JavaScript provides a function Object.freeze to prevent data mutation.

Once the object is frozen, you can no longer add, update, or delete properties from it. Any attempt at changing the object will be rejected without an error.

let obj = {  
  name:"FreeCodeCamp"  
  review:"Awesome"  
};  
Object.freeze(obj);  
obj.review = "bad"; //will be ignored. Mutation not allowed

## Use Arrow Functions to Write Concise Anonymous Functions

In JavaScript, we often don't need to name our functions, especially when passing a function as an argument to another function. Instead, we create inline functions. We don't need to name these functions because we do not reuse them anywhere else.

To achieve this, we often use the following syntax:

const myFunc = function() {  
  const myVar = "value";  
  return myVar;  
}

When there is no function body, and only a return value, arrow function syntax allows you to omit the keyword return as well as the brackets surrounding the code. This helps simplify smaller functions into one-line statements:

const myFunc= () => "value"

This code will still return value by default.

In JS, we often, we often don't name our functions. Callback functions, for example.

Because we often create function just to pass it as argument to some other function, and we might not need to reuse this function anywhere.

To achieve this, we often use the boilerplate

function(){

// function body

return ;

}

ES6 provides you with the syntactic sugar to not having to write this. Instead, you can use arrow function syntax

() => (  
// function body  
return ;  
)

When there is no function body, and only a return value, you can simply write that as a one-liner

```js

() => (/\* return value \*/)

## Set Default Parameters for Your Functions

In order to help us create more flexible functions, ES6 introduces *default parameters*for functions.

Check out this code:

function greeting(name = "Anonymous") {  
  return "Hello " + name;  
}  
console.log(greeting("John")); // Hello John  
console.log(greeting()); // Hello Anonymous

The default parameter kicks in when the argument is not specified (it is undefined). As you can see in the example above, the parameter namewill receive its default value "Anonymous"when you do not provide a value for the parameter. You can add default values for as many parameters as you want.

## Use the Rest Operator with Function Parameters

In order to help us create more flexible functions, ES6 introduces the rest operator for function parameters. With the rest operator, you can create functions that take a variable number of arguments. These arguments are stored in an array that can be accessed later from inside the function.

Rest: Collects multiple elements and condenses into a single array element.

Check out this code:

function howMany(...args) {  
  return "You have passed " + args.length + " arguments.";  
}  
console.log(howMany(0, 1, 2)); // You have passed 3 arguments  
console.log(howMany("string", null, [1, 2, 3], { })); // You have passed 4 arguments.

The rest operator eliminates the need to check the args array and allows us to apply map(), filter()and reduce()on the parameters array.

Spread Operator

<https://www.youtube.com/watch?feature=player_embedded&v=iLx4ma8ZqvQ>

Spread: It is … Allows expressions to be expanded in places where multiple arguments, elements, or

However, the spread operator only works in-place, like in an argument to a function or in an array literal.

-Common uses cases:

* Add elements of an existing array into an arrayPass elements (certification and certtoadd example where individual strings were added instead of array within an array).
* Pass elements of an array as arguments to a function.
* variables are expected. It spreads/expands ana array into its elements

## Use Destructuring Assignment to Assign Variables from Objects

We saw earlier how spread operator can effectively spread, or unpack, the contents of the array. We can do something similar with objects as well. *Destructuring assignment* is special syntax for neatly assigning values taken directly from an object to variables.

Consider the following ES5 code:

var voxel = {x: 3.6, y: 7.4, z: 6.54 };  
var x = voxel.x; // x = 3.6  
var y = voxel.y; // y = 7.4  
var z = voxel.z; // z = 6.54

Here's the same assignment statement with ES6 destructuring syntax:

const {x, y, z } = voxel; // x = 3.6, y = 7.4, z = 6.54

If instead you want to store the values of voxel.xinto a, voxel.yinto b, and voxel.zinto c, you have that freedom as well.

const {x : a, y : b, z : c } = voxel // a = 3.6, b = 7.4, c = 6.54

You may read it as "get the field x and copy the value into a," and so on.

## ES6: Use Destructuring Assignment to Assign Variables from Nested Objects

We can similarly destructure *nested* objects into variables.

Consider the following code:

const a = {  
  start: { x: 5, y: 6},  
  end: { x: 6, y: -9 }  
};  
const { start : { x: startX, y: startY }} = a;  
console.log(startX, startY); // 5, 6

In the example above, the variable start is assigned the value of a.start, which is also an object.

## se Destructuring Assignment to Assign Variables from Arrays

ES6 makes destructuring arrays as easy as destructuring objects.

One key difference between the spread operator and array destructuring is that the spread operator unpacks all contents of an array into a comma-separated list. Consequently, you cannot pick or choose which elements you want to assign to variables.

Destructuring an array lets us do exactly that:

const [a, b] = [1, 2, 3, 4, 5, 6];  
console.log(a, b); // 1, 2

## ES6: Use Destructuring Assignment with the Rest Operator to Reassign Array Elements

In some situations involving array destructuring, we might want to collect the rest of the elements into a separate array.

The result is similar to Array.prototype.slice(), as shown below:

const [a, b, ...arr] = [1, 2, 3, 4, 5, 7];  
console.log(a, b); // 1, 2  
console.log(arr); // [3, 4, 5, 7]

Variables aand btake the first and second values from the array. After that, because of rest operator's presence, arr gets rest of the values in the form of an array.

## ES6: Write Concise Declarative Functions with ES6

When defining functions within objects in ES5, we have to use the keyword functionas follows:

const person = {  
  name: "Taylor",  
  sayHello: function() {  
    return `Hello! My name is ${this.name}.`;  
  }  
};

With ES6, You can remove the functionkeyword and colon altogether when defining functions in objects. Here's an example of this syntax:

const person = {  
  name: "Taylor",  
  sayHello() {  
    return `Hello! My name is ${this.name}.`;  
  }  
};

## ES6: Understand the Differences Between import and require

In the past, the function require()would be used to import the functions and code in external files and modules. While handy, this presents a problem: some files and modules are rather large, and you may only need certain code from those external resources.

ES6 gives us a very handy tool known as import. With it, we can choose which parts of a module or file to load into a given file, saving time and memory.

Consider the following example. Imagine that math\_array\_functionshas about 20 functions, but I only need one, countItems, in my current file. The old require()approach would force me to bring in all 20 functions. With this new importsyntax, I can bring in just the desired function, like so:

import { countItems } from "math\_array\_functions"

A description of the above code:

import { function } from "file\_path\_goes\_here"  
// We can also import variables the same way!

There are a few ways to write an import statement, but the above is a very common use-case.

**Note**  
The whitespace surrounding the function inside the curly braces is a best practice - it makes it easier to read the importstatement.

**Note**  
The lessons in this section handle non-browser features. import, and the statements we introduce in the rest of these lessons, won't work on a browser directly. However, we can use various tools to create code out of this to make it work in browser.

**Note**  
In most cases, the file path requires a ./before it; otherwise, node will look in the node\_modulesdirectory first trying to load it as a dependency.

Add the appropriate import statement that will allow the current file to use the capitalizeStringfunction. The file where this function lives is called "string\_functions", and it is in the same directory as the current file.

## ES6: Use export to Reuse a Code Block

In the previous challenge, you learned about importand how it can be leveraged to import small amounts of code from large files. In order for this to work, though, we must utilize one of the statements that goes with import, known as export. When we want some code - a function, or a variable - to be usable in another file, we must export it in order to import it into another file. Like import, exportis a non-browser feature.

The following is what we refer to as a named export. With this, we can import any code we export into another file with the importsyntax you learned in the last lesson. Here's an example:

const capitalizeString = (string) => {  
  return string.charAt(0).toUpperCase() + string.slice(1);  
}  
export { capitalizeString } //How to export functions.  
export const foo = "bar"; //How to export variables.

## ES6: Use \* to Import Everything from a File

Suppose you have a file that you wish to import all of its contents into the current file. This can be done with the import \* syntax.

Here's an example where the contents of a file named "math\_functions"are imported into a file in the same directory:

import \* as myMathModule from "math\_functions";  
myMathModule.add(2,3);  
myMathModule.subtract(5,3);

And breaking down that code:

import \* as object\_with\_name\_of\_your\_choice from "file\_path\_goes\_here"  
object\_with\_name\_of\_your\_choice.imported\_function

You may use any name following the import \* as portion of the statement. In order to utilize this method, it requires an object that receives the imported values. From here, you will use the dot notation to call your imported values.

## ES6: Create an Export Fallback with export default

In the exportlesson, you learned about the syntax referred to as a named export. This allowed you to make multiple functions and variables available for use in other files.

There is another exportsyntax you need to know, known as export default. Usually you will use this syntax if only one value is being exported from a file. It is also used to create a fallback value for a file or module.

Here is a quick example of export default:

export default function add(x,y) {  
  return x + y;  
}

Note: Since export defaultis used to declare a fallback value for a module or file, you can only have one value be a default export in each module or file. Additionally, you cannot use export default with var, let, or const

## ES6: Import a Default Export

In the last challenge, you learned about export defaultand its uses. It is important to note that, to import a default export, you need to use a different importsyntax.

In the following example, we have a function, add, that is the default export of a file, "math\_functions". Here is how to import it:

import add from "math\_functions";  
add(5,4); //Will return 9

The syntax differs in one key place - the imported value, add, is not surrounded by curly braces, {}. Unlike exported values, the primary method of importing a default export is to simply write the value's name after import.

## Introduction to the Regular Expression Challenges

Regular expressions are special strings that represent a search pattern. Also known as "regex" or "regexp", they help programmers match, search, and replace text. Regular expressions can appear cryptic because a few characters have special meaning. The goal is to combine the symbols and text into a pattern that matches what you want, but only what you want. This section will cover the characters, a few shortcuts, and the common uses for writing regular expressions.

* 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 trueor falseif your pattern finds something or not.

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

* 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.
* 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/.
* 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 iflag. 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".

## Regular Expressions: 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/);  
// Returns ["Hello"]  
let ourStr = "Regular expressions";  
let ourRegex = /expressions/;  
ourStr.match(ourRegex);  
// Returns ["expressions"]

**Finding more than the FIrest mAtch**

To search or extract a pattern more than once, you can use the gflag.

let repeatRegex = /Repeat/g;  
testStr.match(repeatRegex);  
// Returns ["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.

Greedy vs Lazy Match

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"].

## 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 caretcharacter (^) inside a character setto create a negated character set in the form [^thingsThatWillNotBeMatched]. Outside of a character set, the caretis 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);  
// Returns true  
let notFirst = "You can't find Ricky now.";  
firstRegex.test(notFirst);  
// Returns false

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.

LOOKAHEADS

Lookaheadsare 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.

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); // Returns true

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 \1to 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); // Returns true  
repeatStr.match(repeatRegex); // Returns ["regex regex", "regex"]

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

## 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");  
// Returns "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');  
// Returns "Camp Code"

REMOVE WHITESPACE

Sometimes whitespace characters around strings are not wanted but are there. Typical processing of strings is to remove the whitespace at the start and end of it.

## Debugging: DevTools

Typeof to check the data structure, or type, of a variable. This is useful in debugging when working with multiple data types. Examples:

* Console.log(typeof “”) //output “string”
* Console.log( typeof [] ) //outputs “object”

JavaScript recognizes six primitive (immutable) data types: Boolean, Null, Undefined, Number, String, and Symbol (new with ES6) and one type for mutable items: Object. Note that in JavaScript, arrays are technically a type of object.

Types of syntax errors:

* Misspelled variables eg revvable and receivable
* Another syntax error to be aware of is that all opening parentheses, brackets, curly braces, and quotes have a closing pair.
* Almost every value on its own in JavaScript evaluates to true, except what are known as the "falsy" values:
  + False
  + 0
  + ""(an empty string)
  + ,NaN
  + Undefined
  + and null.
* When a function or method doesn't take any arguments, you may forget to include the (empty) opening and closing parentheses when calling it. Often times the result of a function call is saved in a variable for other use in your code. This error can be detected by logging variable values (or their types) to the console and seeing that one is set to a function reference, instead of the expected value the function returns.
* The variables in the following example are different:

function myFunction() {  
  return "You rock!";  
}  
let varOne = myFunction; // set to equal a function  
let varTwo = myFunction(); // set to equal the string "You rock!"

* Fix the code so the variable result is set to the value returned from calling the function getNine.
* Original: let result = getNine; Solution: let result = getNine()

## Catch Arguments Passed in the Wrong Order When Calling a Function

Continuing the discussion on calling functions, the next bug to watch out for is when a function's arguments are supplied in the incorrect order. If the arguments are different types, such as a function expecting an array and an integer, this will likely throw a runtime error. If the arguments are the same type (all integers, for example), then the logic of the code won't make sense. Make sure to supply all required arguments, in the proper order to avoid these issues.

## Catch Off By One Errors When Using Indexing

Off by one errors (sometimes called OBOE) crop up when you're trying to target a specific index of a string or array (to slice or access a segment), or when looping over the indices of them.

let alphabet = "abcdefghijklmnopqrstuvwxyz";  
let len = alphabet.length;  
for (let i = 0; i <= len; i++) {  
  // loops one too many times at the end  
  console.log(alphabet[i]);  
}  
for (let j = 1; j < len; j++) {  
  // loops one too few times and misses the first character at index 0  
  console.log(alphabet[j]);  
}  
for (let k = 0; k < len; k++) {  
  // Goldilocks approves - this is just right  
  console.log(alphabet[k]);  
}

**Basic Data Structures**

The key difference between pop()and shift()and their cousins push()and unshift(), is that neither method takes parameters, and each only allows an array to be modified by a single element at a time.

Let's take a look:

let greetings = ['whats up?', 'hello', 'see ya!'];  
  
greetings.pop();  
// now equals ['whats up?', 'hello']  
  
greetings.shift();  
// now equals ['hello']

We can also return the value of the removed element with either method like this:

let popped = greetings.pop();  
// returns 'hello'  
// greetings now equals []

SPLICE

splice()can take up to 3 parameters, but for now, we'll focus on just the first 2. The first two parameters of splice()are integers which represent indexes, or positions, of the array that splice()is being called upon.

splice()'s first parameter represents the index on the array from which to begin removing elements, while the second parameter indicates the number of elements to delete. For example:

let array = ['today', 'was', 'not', 'so', 'great'];  
  
array.splice(2, 2);  
// remove 2 elements beginning with the 3rd element  
// array now equals ['today', 'was', 'great']

In addition to removing elements, we can use that third parameter, which represents one or more elements, to *add* them as well. This can be incredibly useful for quickly switching out an element, or a set of elements, for another.

function colorChange(arr, index, newColor) {  
  arr.splice(index, 1, newColor);  
  return arr;  
}

SLICE

The next method we will cover is slice(). slice(), rather than modifying an array, copies, or *extracts*, a given number of elements to a new array, leaving the array it is called upon untouched. slice()takes only 2 parameters — the first is the index at which to begin extraction, and the second is the index at which to stop extraction (*extraction will occur up to, but not including the element at this index*). Consider this:

let weatherConditions = ['rain', 'snow', 'sleet', 'hail', 'clear'];  
  
let todaysWeather = weatherConditions.slice(1, 3);  
// todaysWeather equals ['snow', 'sleet'];  
// weatherConditions still equals ['rain', 'snow', 'sleet', 'hail', 'clear']

In effect, we have created a new array by extracting elements from an existing array.

SPREAD OPERATOR

While slice() allows us to be selective about what elements of an array to copy, among several other useful tasks, ES6's new *spread operator* allows us to easily copy *all* of an array's elements, in order, with a simple and highly readable syntax. The spread syntax simply looks like this: ...

## Combine Arrays with the Spread Operator

Another huge advantage of the spread operator, is the ability to combine arrays, or to insert all the elements of one array into another, at any index. With more traditional syntaxes, we can concatenate arrays, but this only allows us to combine arrays at the end of one, and at the start of another. Spread syntax makes the following operation extremely simple:

let thisArray = ['sage', 'rosemary', 'parsley', 'thyme'];  
  
let thatArray = ['basil', 'cilantro', ...thisArray, 'coriander'];  
// thatArray now equals ['basil', 'cilantro', 'sage', 'rosemary', 'parsley', 'thyme', 'coriander']

Using spread syntax, we have just achieved an operation that would have been more more complex and more verbose had we used traditional methods.

## Check For The Presence of an Element With indexOf()

Since arrays can be changed, or mutated, at any time, there's no guarantee about where a particular piece of data will be on a given array, or if that element even still exists. Luckily, JavaScript provides us with another built-in method, indexOf(), that allows us to quickly and easily check for the presence of an element on an array. indexOf()takes an element as a parameter, and when called, it returns the position, or index, of that element, or -1if the element does not exist on the array.

For example:

let fruits = ['apples', 'pears', 'oranges', 'peaches', 'pears'];  
  
fruits.indexOf('dates') // returns -1  
fruits.indexOf('oranges') // returns 2  
fruits.indexOf('pears') // returns 1, the first index at which the element exists

## Iterate Through All an Array's Items Using For Loops

Sometimes when working with arrays, it is very handy to be able to iterate through each item to find one or more elements that we might need, or to manipulate an array based on which data items meet a certain set of criteria. JavaScript offers several built in methods that each iterate over arrays in slightly different ways to achieve different results (such as every(), forEach(), map(), etc.), however the technique which is most flexible and offers us the greatest amount of control is a simple forloop.

Consider the following:

function greaterThanTen(arr) {  
  let newArr = [];  
  for (let i = 0; i < arr.length; i++) {  
    if (arr[i] > 10) {  
      newArr.push(arr[i]);  
    }  
  }  
  return newArr;  
}  
  
greaterThanTen([2, 12, 8, 14, 80, 0, 1]);  
// returns [12, 14, 80]

Using a for loop, this function iterates through and accesses each element of the array, and subjects it to a simple test that we have created. In this way, we have easily and programmatically determined which data items are greater than 10, and returned a new array containing those items.

**OBJECTS**

Now you know what objects are and their basic features and advantages. In short, they are key-value stores which provide a flexible, intuitive way to structure data, ***and***, they provide very fast lookup time. Throughout the rest of these challenges, we will describe several common operations you can perform on objects so you can become comfortable applying these useful data structures in your programs.

Check if an Object Has a Property

Now we can add, modify, and remove keys from objects. But what if we just wanted to know if an object has a specific property? JavaScript provides us with two different ways to do this. One uses the hasOwnProperty() method and the other uses the in keyword. If we have an object userswith a property of Alan, we could check for its presence in either of the following ways:

users.hasOwnProperty('Alan');  
'Alan' in users;  
// both return true

Iterate through the Keys of an Object

Sometimes you may need to iterate through all the keys within an object. This requires a specific syntax in JavaScript called a *for...in* statement. For our usersobject, this could look like:

for (let user in users) {  
  console.log(user);  
};  
  
// logs:  
Alan  
Jeff  
Sarah  
Ryan

In this statement, we defined a variable user, and as you can see, this variable was reset during each iteration to each of the object's keys as the statement looped through the object, resulting in each user's name being printed to the console.

**NOTE:**

Objects do not maintain an ordering to stored keys like arrays do; thus a keys position on an object, or the relative order in which it appears, is irrelevant when referencing or accessing that key.

## Generate an Array of All Object Keys with Object.keys()

We can also generate an array which contains all the keys stored in an object using the Object.keys() method and passing in an object as the argument. This will return an array with strings representing each property in the object. Again, there will be no specific order to the entries in the array.

Now you've seen all the basic operations for JavaScript objects. You can add, modify, and remove key-value pairs, check if keys exist, and iterate over all the keys in an object. As you continue learning JavaScript you will see even more versatile applications of objects. Additionally, the optional Advanced Data Structures lessons later in the curriculum also cover the ES6 *Map* and *Set*objects, both of which are similar to ordinary objects but provide some additional features. Now that you've learned the basics of arrays and objects, you're fully prepared to begin tackling more complex problems using JavaScript!

## Basic Algorithm Scripting:

## Problem Explanation:

Return the factorial of the provided integer. If the integer is represented with the letter n, a factorial is the product of all positive integers less than or equal to n.

Factorials are often represented with the shorthand notation n!

For example: 5! = 1 \* 2 \* 3 \* 4 \* 5 = 120

## Hint: 1

This one starts easily since 0! = 1, so you can go ahead and simply return 1 there.

We can use that as an if in order to break the loop we’re going to create using a **recursive function**. It will check if the number you gave the function is 0 (which would be the end of your factorial chain). Functions “end” when they return anything. In fact, **all** functions without an explicit return statement will return undefined.

This is also why **instead** of having “finished”, a function is always said to “have returned”. And now this…

try to solve the problem now

## Hint: 2

**Understanding recursion**

Recursion refers to a function repeating (calling) itself. In this case we are basically returning the given number (i.e. 5), multiplied by the function itself but this time the value passed to the num parameter is num-1 (which initially translates to 4). The very function is going to **run inside itself** interesting, eh?

try to solve the problem now

## Hint: 3

**Understanding the flow**

The first **returned** value can be visualized better if you think about those parenthesis operations you did in secondary school where you do the math inside every parenthesis from inside out, bracket and square bracket until you get a final result (a total). This time it’s the same thing, look at the program flow:

### During the first execution of the function:

[**num** = 5]

Is 5 equal to 1 or 0? **No** ---> Oki doki, let’s continue…

**Returns:**

(**5** (second execution: ***4***\_(third execution: ***3*** \_(fourth execution\_: **2** \_fifth execution: **1**))))

What it returns can be viewed as (5\*(4\*(3\*(2\*1)))) or just 5 \* 4 \* 3 \* 2 \* 1, and the function will return the result of that operation: 120. Now, let’s check what the rest of the executions do:

### During the rest of the executions:

**Second Execution**: num = 5-1 = **4** -> is num 0 or 1? No

## Code Solution:

function factorialize(num) {

if (num === 0) { return 1; }

return num \* factorialize(num-1);

}

factorialize(5);

## Introduction to the Object Oriented Programming Challenges

At its core, software development solves a problem or achieves a result with computation. The software development process first defines a problem, then presents a solution. Object oriented programming is one of several major approaches to the software development process.  
  
As its name implies, object oriented programming organizes code into object definitions. These are sometimes called classes, and they group together data with related behavior. The data is an object's attributes, and the behavior (or functions) are methods.  
  
The object structure makes it flexible within a program. Objects can transfer information by calling and passing data to another object's methods. Also, new classes can receive, or inherit, all the features from a base or parent class. This helps to reduce repeated code.  
  
Your choice of programming approach depends on a few factors. These include the type of problem, as well as how you want to structure your data and algorithms. This section covers object oriented programming principles in JavaScript.

## Object Oriented Programming: Make Code More Reusable with the this Keyword

The last challenge introduced a method to the duck object. It used duck.name dot notation to access the value for the name property within the return statement:

sayName: function() {return "The name of this duck is " + duck.name + ".";}

While this is a valid way to access the object's property, there is a pitfall here. If the variable name changes, any code referencing the original name would need to be updated as well. In a short object definition, it isn't a problem, but if an object has many references to its properties there is a greater chance for error.

A way to avoid these issues is with the this keyword:

This is a deep topic, and the above example is only one way to use it. In the current context, this refers to the object that the method is associated with: duck.

If the object's name is changed to mallard, it is not necessary to find all the references to duck in the code. It makes the code reusable and easier to read.

CONSTRUCTORS

Constructors are functions that create new objects. They define properties and behaviors that will belong to the new object. Think of them as a blueprint for the creation of new objects.

Here is an example of a constructor:

function Bird() {  
  this.name = "Albert";  
  this.color = "blue";  
  this.numLegs = 2;  
}

This constructor defines a Bird object with properties name, color, and numLegss et to Albert, blue, and 2, respectively.

Constructors follow a few conventions:

* Constructors are defined with a capitalized name to distinguish them from other functions that are not constructors.
* Constructors use the keyword this to set properties of the object they will create. Inside the constructor, this refers to the new object it will create.
* Constructors define properties and behaviors instead of returning a value as other functions might.

Here's the Bird constructor from the previous challenge:

* function Bird() {  
    this.name = "Albert";  
    this.color = "blue";  
    this.numLegs = 2;  
    // "this" inside the constructor always refers to the object being created  
  }  
    
  let blueBird = new Bird();
* Notice that the new operator is used when calling a constructor. This tells JavaScript to create a new instance of Bird called blueBird. Without the new operator, thisinside the constructor would not point to the newly created object, giving unexpected results.
* Now blueBirdhas all the properties defined inside the Birdconstructor:
* blueBird.name; // => Albert  
  blueBird.color; // => blue  
  blueBird.numLegs; // => 2

## Extend Constructors to Receive Arguments

Suppose you were writing a program to keep track of hundreds or even thousands of different birds in an aviary. It would take a lot of time to create all the birds, then change the properties to different values for every one.

To more easily create different Bird objects, you can design your Bird constructor to accept parameters:

function Bird(name, color) {  
  this.name = name;  
  this.color = color;  
  this.numLegs = 2;  
}

Then pass in the values as arguments to define each unique bird into the Birdconstructor:

let cardinal = new Bird("Bruce", "red");

This gives a new instance of Bird with name and color properties set to Bruce and red, respectively. The numLegs property is still set to 2.

The cardinal has these properties:

cardinal.name // => Bruce  
cardinal.color // => red  
cardinal.numLegs // => 2

The constructor is more flexible. It's now possible to define the properties for each Bird at the time it is created, which is one way that JavaScript constructors are so useful. They group objects together based on shared characteristics and behavior and define a blueprint that automates their creation.

INSTANCEOF

Anytime a constructor function creates a new object, that object is said to be an instance of its constructor. JavaScript gives a convenient way to verify this with the instanceof operator. Instanceof allows you to compare an object to a constructor, returning true or false based on whether or not that object was created with the constructor. Here's an example:

let Bird = function(name, color) {  
  this.name = name;  
  this.color = color;  
  this.numLegs = 2;  
}  
  
let crow = new Bird("Alexis", "black");  
  
crow instanceof Bird; // => true

## Understand Own Properties

In the following example, the Bird constructor defines two properties: name and numLegs:

function Bird(name) {  
  this.name = name;  
  this.numLegs = 2;  
}  
  
let duck = new Bird("Donald");  
let canary = new Bird("Tweety");

name and numLegs are called own properties, because they are defined directly on the instance object. That means that duck and canary each has its own separate copy of these properties.

In fact every instance of Bird will have its own copy of these properties.

The following code adds all of the own properties of duck to the array ownProps:

let ownProps = [];  
  
for (let property in duck) {  
  if(duck.hasOwnProperty(property)) {  
    ownProps.push(property);  
  }  
}  
  
console.log(ownProps); // prints [ "name", "numLegs" ]

## Use Prototype Properties to Reduce Duplicate Code

Since numLegs will probably have the same value for all instances of Bird, you essentially have a duplicated variable numLegs inside each Bird instance.

This may not be an issue when there are only two instances, but imagine if there are millions of instances. That would be a lot of duplicated variables.

A better way is to use Bird’s prototype. The prototype is an object that is shared among ALL instances of Bird. Here's how to add numLegs to the Bird prototype:

Bird.prototype.numLegs = 2;

Now all instances of Bird have the numLegs property.

console.log(duck.numLegs); // prints 2  
console.log(canary.numLegs); // prints 2

Since all instances automatically have the properties on the prototype, think of a prototype as a "recipe" for creating objects.

Note that the prototype for duck and canary is part of the Bird constructor as Bird.prototype. Nearly every object in JavaScript has a prototype property which is part of the constructor function that created it.

## MASS UPDATE BY CHANGING PROTOTYPE TO AN OBJECT

Up until now you have been adding properties to the prototype individually:

Bird.prototype.numLegs = 2;

This becomes tedious after more than a few properties. A more efficient way is to set the prototype to a new object that already contains the properties. This way, the properties are added all at once:

 Also, whenever a prototype is manually set to a new object, remember to define the constructor property:

Bird.prototype = {  
  constructor: Bird, // define the constructor property  
  numLegs: 2,  
  eat: function() {  
    console.log("nom nom nom");  
  },  
  describe: function() {  
    console.log("My name is " + this.name);   
  }  
};

## INHERITING PROTOTYPES

Just like people inherit genes from their parents, an object inherits its prototype directly from the constructor function that created it. For example, here the Bird constructor creates the duck object:

function Bird(name) {  
  this.name = name;  
}  
  
let duck = new Bird("Donald");

Duck inherits its prototype from the Bird constructor function. You can show this relationship with the isPrototypeOf method:

Bird.prototype.isPrototypeOf(duck);  
// returns true

## Use Inheritance So You Don't Repeat Yourself

There's a principle in programming called Don't Repeat Yourself (DRY). The reason repeated code is a problem is because any change requires fixing code in multiple places. This usually means more work for programmers and more room for errors.

The code can be edited to follow the DRY principle by creating a supertype(or parent) called Animal:

function Animal() { };  
  
Animal.prototype = {  
  constructor: Animal,   
  describe: function() {  
    console.log("My name is " + this.name);  
  }  
};

Since Animal includes the describe method, you can remove it from Bird and Dog:

## INHERIT BEHAVIORS FROM A SUPERTYPE

This and the next challenge will cover how to reuse Animal's methods inside Bird and Dog without defining them again. It uses a technique called inheritance.

This challenge covers the first step: make an instance of the supertype(or parent).

You already know one way to create an instance of Animal using the new operator:

let animal = new Animal();

There are some disadvantages when using this syntax for inheritance, which are too complex for the scope of this challenge. Instead, here's an alternative approach without those disadvantages:

let animal = Object.create(Animal.prototype);

Object.create(obj)creates a new object, and sets obj as the new object's prototype. Recall that the prototype is like the "recipe" for creating an object. By setting the prototype of animal to be Animal'sprototype, you are effectively giving the animal instance the same "recipe" as any other instance of Animal.

## 2nd Step:

This challenge covers the next step: set the prototype of the subtype(or child)—in this case, Bird—to be an instance of Animal.

Bird.prototype = Object.create(Animal.prototype);

Remember that the prototype is like the "recipe" for creating an object. In a way, the recipe for Birdnow includes all the key "ingredients" from Animal.

let duck = new Bird("Donald");  
duck.eat(); // prints "nom nom nom"

duck inherits all of Animal's properties, including the eat method.

## RESET INHERITED CONTRCUTOR PROPERTY

Birdshould show that they were constructed by Bird and not Animal. To do so, you can manually set Bird's constructor property to the Bird object:

Bird.prototype.constructor = Bird;  
duck.constructor // function Bird(){...}

## Add Methods After Inheritance

A constructor function that inherits its prototype object from a supertype constructor function can still have its own methods in addition to inherited methods.

For example, Bird is a constructor that inherits its prototype from Animal:

function Animal() { }  
Animal.prototype.eat = function() {  
  console.log("nom nom nom");  
};  
function Bird() { }  
Bird.prototype = Object.create(Animal.prototype);  
Bird.prototype.constructor = Bird;

In addition to what is inherited from Animal, you want to add behavior that is unique to Birdobjects. Here, Bird will get a fly()function. Functions are added to Bird'sprototypethe same way as any constructor function:

Bird.prototype.fly = function() {  
  console.log("I'm flying!");  
};

Now instances of Bird will have both eat() and fly() methods:

let duck = new Bird();  
duck.eat(); // prints "nom nom nom"  
duck.fly(); // prints "I'm flying!"

## Override Inherited Methods

In previous lessons, you learned that an object can inherit its behavior (methods) from another object by cloning its prototype object:

ChildObject.prototype = Object.create(ParentObject.prototype);

Then the ChildObjectreceived its own methods by chaining them onto its prototype:

ChildObject.prototype.methodName = function() {...};

It's possible to override an inherited method. It's done the same way - by adding a method to ChildObject.prototypeusing the same method name as the one to override.

Here's an example of Birdoverriding the eat()method inherited from Animal:

function Animal() { }  
Animal.prototype.eat = function() {  
  return "nom nom nom";  
};  
function Bird() { }  
  
// Inherit all methods from Animal  
Bird.prototype = Object.create(Animal.prototype);  
  
// Bird.eat() overrides Animal.eat()  
Bird.prototype.eat = function() {  
  return "peck peck peck";  
};

## Use a Mixin to Add Common Behavior Between Unrelated Objects

As you have seen, behavior is shared through inheritance. However, there are cases when inheritance is not the best solution. Inheritance does not work well for unrelated objects like Bird and Airplane. They can both fly, but a Bird is not a type of Airplane and vice versa.

For unrelated objects, it's better to use mixins. A mixin allows other objects to use a collection of functions.

let flyMixin = function(obj) {  
  obj.fly = function() {  
    console.log("Flying, wooosh!");  
  }  
};

The flyMixin takes any object and gives it the fly method.

let bird = {  
  name: "Donald",  
  numLegs: 2  
};  
  
let plane = {  
  model: "777",  
  numPassengers: 524  
};  
  
flyMixin(bird);  
flyMixin(plane);

Here birdand planeare passed into flyMixin, which then assigns the fly function to each object. Now bird and plane can both fly:

bird.fly(); // prints "Flying, wooosh!"  
plane.fly(); // prints "Flying, wooosh!"

Note how the mixin allows for the same fly method to be reused by unrelated objects bird and plane.

## Use Closure to Protect Properties Within an Object from Being Modified Externally

In the previous challenge, bird had a public property name. It is considered public because it can be accessed and changed outside of bird's definition.

bird.name = "Duffy";

Therefore, any part of your code can easily change the name of bird to any value. Think about things like passwords and bank accounts being easily changeable by any part of your codebase. That could cause a lot of issues.

The simplest way to make properties private is by creating a variable within the constructor function. This changes the scope of that variable to be within the constructor function versus available globally. This way, the property can only be accessed and changed by methods also within the constructor function.

function Bird() {  
  let hatchedEgg = 10; // private property  
  
  this.getHatchedEggCount = function() { // publicly available method that a bird object can use  
    return hatchedEgg;  
  };  
}  
let ducky = new Bird();  
ducky.getHatchedEggCount(); // returns 10

Here getHachedEggCountis a privileged method, because it has access to the private variable hatchedEgg. This is possible because hatchedEgg is declared in the same context as getHachedEggCount. In JavaScript, a function always has access to the context in which it was created. This is called closure.

## Understand the Immediately Invoked Function Expression (IIFE)

A common pattern in JavaScript is to execute a function as soon as it is declared:

(function () {  
  console.log("Chirp, chirp!");  
})(); // this is an anonymous function expression that executes right away  
// Outputs "Chirp, chirp!" immediately

Note that the function has no name and is not stored in a variable. The two parentheses () at the end of the function expression cause it to be immediately executed or invoked. This pattern is known as an immediately invoked function expression or IIFE.

## Use an IIFE to Create a Module

An immediately invoked function expression(IIFE) is often used to group related functionality into a single object or module. For example, an earlier challenge defined two mixins:

function glideMixin(obj) {  
  obj.glide = function() {  
    console.log("Gliding on the water");  
  };  
}  
function flyMixin(obj) {  
  obj.fly = function() {  
    console.log("Flying, wooosh!");  
  };  
}

We can group these mixins into a module as follows:

let motionModule = (function () {  
  return {  
    glideMixin: function (obj) {  
      obj.glide = function() {  
        console.log("Gliding on the water");  
      };  
    },  
    flyMixin: function(obj) {  
      obj.fly = function() {  
        console.log("Flying, wooosh!");  
      };  
    }  
  }  
}) (); // The two parentheses cause the function to be immediately invoked

Note that you have an immediately invoked function expression(IIFE) that returns an object motionModule. This returned object contains all of the mixinbehaviors as properties of the object.

The advantage of the modulepattern is that all of the motion behaviors can be packaged into a single object that can then be used by other parts of your code. Here is an example using it:

motionModule.glideMixin(duck);  
duck.glide();

## Functional Programming: Learn About Functional Programming

Functional programming is a style of programming where solutions are simple, isolated functions, without any side effects outside of the function scope.

INPUT -> PROCESS -> OUTPUT

Functional programming is about:

1) Isolated functions - there is no dependence on the state of the program, which includes global variables that are subject to change

2) Pure functions - the same input always gives the same output

3) Functions with limited side effects - any changes, or mutations, to the state of the program outside the function are carefully controlled

TERMINOLOGY

Callbacks are the functions that are slipped or passed into another function to decide the invocation of that function. You may have seen them passed to other methods, for example in filter, the callback function tells JavaScript the criteria for how to filter an array.

Functions that can be assigned to a variable, passed into another function, or returned from another function just like any other normal value, are called first class functions. In JavaScript, all functions are first class functions.

The functions that take a function as an argument, or return a function as a return value are called higher order functions.

When the functions are passed in to another function or returned from another function, then those functions which gets passed in or returned can be called a lambda.

## Understand the Hazards of Using Imperative Code

Functional programming is a good habit. It keeps your code easy to manage, and saves you from sneaky bugs. But before we get there, let's look at an imperative approach to programming to highlight where you may have issues.

In English (and many other languages), the imperative tense is used to give commands. Similarly, an imperative style in programming is one that gives the computer a set of statements to perform a task.

Often the statements change the state of the program, like updating global variables. A classic example is writing a for loop that gives exact directions to iterate over the indices of an array.

In contrast, functional programming is a form of declarative programming. You tell the computer what you want done by calling a method or function.

Functional programming is declarative, the original data must nor be changed, whereas most programming is imperative.

* declarative
  + telling the computer **what** you want done.
  + the functions you call handle all the steps to get it done.
* imperative
  + telling the computer **how** you want it done.
  + this means you have to write all the steps that are required.

Knowing this, are there any methods on the array object that you can use? Let those functions do all the work of combining and filtering data.

JavaScript offers many predefined methods that handle common tasks so you don't need to write out how the computer should perform them. For example, instead of using the for loop mentioned above, you could call the map method which handles the details of iterating over an array. This helps to avoid semantic errors, like the "Off By One Errors" that were covered in the Debugging section.

If you haven't already figured it out, the issue in the previous challenge was with the splice call in the tabClose() function. Unfortunately, splice changes the original array it is called on, so the second call to it used a modified array, and gave unexpected results.

This is a small example of a much larger pattern - you call a function on a variable, array, or an object, and the function changes the variable or something in the object.

One of the core principle of functional programming is to not change things. Changes lead to bugs. It's easier to prevent bugs knowing that your functions don't change anything, including the function arguments or any global variable.

The previous example didn't have any complicated operations but the splice method changed the original array, and resulted in a bug.

Recall that in functional programming, changing or altering things is called mutation, and the outcome is called a side effect. A function, ideally, should be a pure function, meaning that it does not cause any side effects.

PASSING ARGUMENTS TO AVOID ECTERNAL DEPENDENCIES

Another principle of functional programming is to always declare your dependencies explicitly. This means if a function depends on a variable or object being present, then pass that variable or object directly into the function as an argument.

There are several good consequences from this principle. The function is easier to test, you know exactly what input it takes, and it won't depend on anything else in your program.

This can give you more confidence when you alter, remove, or add new code. You would know what you can or cannot change and you can see where the potential traps are.

Finally, the function would always produce the same output for the same set of inputs, no matter what part of the code executes it.

## Refactor Global Variables Out of Functions

So far, we have seen two distinct principles for functional programming:

1) Don't alter a variable or object - create new variables and objects and return them if need be from a function.

2) Declare function arguments - any computation inside a function depends only on the arguments, and not on any global object or variable.

MAP METHOD TO EXTRACT DATA FROM AN ARRAY

It would make sense to be able to pass them as arguments to other functions, and return a function from another function. Functions are considered First Class Objects in JavaScript, which means they can be used like any other object. They can be saved in variables, stored in an object, or passed as function arguments.

Let's start with some simple array functions, which are methods on the array object prototype. In this exercise we are looking at Array.prototype.map(), or more simply map.

Remember that the map method is a way to iterate over each item in an array. It creates a new array (without changing the original one) after applying a callback function to every element.

As you have seen from applying Array.prototype.map(), or simply map()earlier, the map method returns an array of the same length as the one it was called on. It also doesn't alter the original array, as long as its callback function doesn't.

In other words, map is a pure function, and its output depends solely on its inputs. Plus, it takes another function as its argument.

It would teach us a lot about map to try to implement a version of it that behaves exactly like the Array.prototype.map()with a for loop or Array.prototype.forEach().

Note: A pure function is allowed to alter local variables defined within its scope, although, it's preferable to avoid that as well.

**CodeAnalogies Explanation of Map, Reduce, Filter** [**here**](https://blog.codeanalogies.com/2018/07/24/javascripts-reduce-method-explained-by-going-on-a-diet/)

Now that we have the combined array, we need to figure out how we are going to extract the correct values. This is where map, filter and reduce come into play.

* [Map](https://blog.codeanalogies.com/2018/02/20/javascript-map-method-explained-by-going-on-a-hike/) lets you transform every element from a beginning array into a final array
* [Filter](https://blog.codeanalogies.com/2018/05/14/javascripts-filter-function-explained-by-applying-to-college/) lets you select only the elements in an array that meet certain criteria
* [Reduce](https://blog.codeanalogies.com/2018/07/24/javascripts-reduce-method-explained-by-going-on-a-diet/) lets you sum up array elements based on certain criteria.

**Implement the Filter Method on a Prototype**

The normal filter method uses a callback function as the 1st parameter. The callback function normally has 1 to 3 arguments, with the 1st argument representing the current element being iterated over. For this challenge, you only have to use this one argument, which is why the function being passed has one argument (item) as seen below:

function(item){

return item % 2 === 1;

}

If you call normal filter method on the s array with the same callback function, you would write:

var s = [23, 65, 98, 5];

console.log(s.filter(function(item){

return item % 2 === 1;

}); // displays [23, 65, 5]

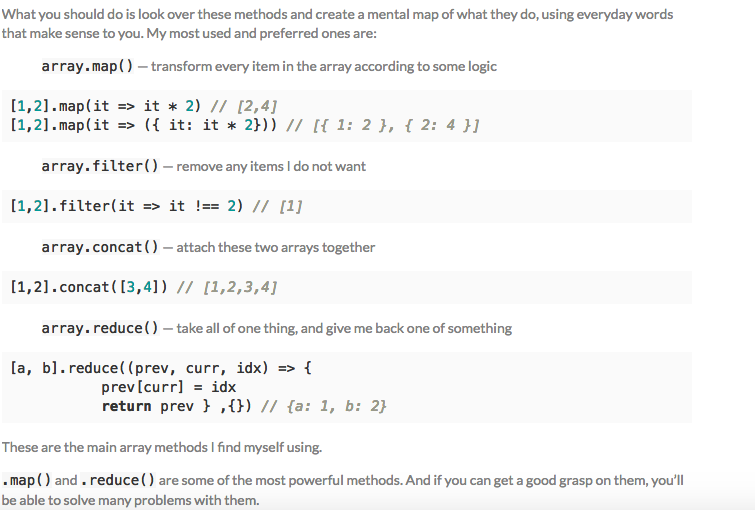
Your custom myFilter method must produce the same result **if** called on the s array. You will use the the loop **of** your choice (**for** loop or forEach method) to iterate through the array and create a **new** array **with** values returned by calling the callback **function** with each array element's value as the argument of the callback function.

REDUCE METHOD: taking an entire array and reducing it to a single value.(Steve Griffith YT video tutorial

Array.prototype.reduce(), or simply reduce(), is the most general of all array operations in JavaScript. You can solve almost any array processing problem using the reducemethod.

This is not the case with the filter and map methods since they do not allow interaction between two different elements of the array. For example, if you want to compare elements of the array, or add them together, filter or map could not process that.

The reduce method allows for more general forms of array processing, and it's possible to show that both filter and map can be derived as a special application of reduce.



Array.Find and .FindIndex()

Example 1:

let arr = ['a','b','c'];

arr.find(k => k=='b');  
// 'b'

Example 2:

let arr = [2, 4, 6, 8, 9, 10, 12];

function isOdd(i) {  
 return i % 2 !== 0;  
}

Finally, we can use Array.find() to pass in our isOdd() function on the arrArray:

arr.find(isOdd);  
// 9

You’ll soon see that findIndex() works very similarly to find(). Instead of returning the value, **findIndex() returns the index of the first element in the array that satisfies the given test.**

* If the test is not satisfied, -1 is returned

#### Example #1

We’ll be using the same examples as above:

let arr = ['a','b','c'];

arr.findIndex(k => k=='b');  
// 1

SOME METHOD

The some method works with arrays to check if *any* element passes a particular test. It returns a Boolean value – true if any of the values meet the criteria, false if not.

For example, the following code would check if any element in the numbersarray is less than 10:

var numbers = [10, 50, 8, 220, 110, 11];  
numbers.some(function(currentValue) {  
  return currentValue < 10;  
});  
// Returns true

## Introduction to Currying and Partial Application

The arity of a function is the number of arguments it requires. Currying a function means to convert a function of N arity into N functions of arity 1.

In other words, it restructures a function so it takes one argument, then returns another function that takes the next argument, and so on.

Here's an example:

//Un-curried function  
function unCurried(x, y) {  
  return x + y;  
}  
  
//Curried function  
function curried(x) {  
  return function(y) {  
    return x + y;  
  }  
}  
curried(1)(2) // Returns 3

This is useful in your program if you can't supply all the arguments to a function at one time. You can save each function call into a variable, which will hold the returned function reference that takes the next argument when it's available. Here's an example using the curriedfunction in the example above:

// Call a curried function in parts:  
var funcForY = curried(1);  
console.log(funcForY(2)); // Prints 3

Similarly, partial application can be described as applying a few arguments to a function at a time and returning another function that is applied to more arguments.

Here's an example:

//Impartial function  
function impartial(x, y, z) {  
  return x + y + z;  
}  
var partialFn = impartial.bind(this, 1, 2);  
partialFn(10); // Returns 13

**Title Case a Sentence With a FOR Loop**

String.toLowerCase() converts a string to lowercase, and String.toUpperCase()converts a string to uppercase.

var text = 'This sentence has some MIXED CASE LeTTeRs in it.';

// returns 'this sentence has some mixed case letters in it.'

var lower = text.toLowerCase();

// returns 'THIS SENTENCE HAS SOME MIXED CASE LETTERS IN IT.'

var upper = text.toUpperCase();

* The **toLowerCase()** method returns the calling string value converted to lowercase
* The **split()** method splits a String object into an array of strings by separating the string into substrings.
* The **charAt()** method returns the specified character from a string.
* The **slice()** method extracts a section of a string and returns a new string.
* The **join()** method joins all elements of an array into a string.