Basic JavaScript

1. Comment Your JavaScript Code

Description

Comments are lines of code that JavaScript will intentionally ignore. Comments are a great way to leave notes to yourself and to other people who will later need to figure out what that code does. There are two ways to write comments in JavaScript: Using `//` will tell JavaScript to ignore the remainder of the text on the current line:

```
// This is an in-line comment.
```

You can make a multi-line comment beginning with `/*` and ending with `*/`:

```
/* This is a
multi-line comment */
```

Best Practice
As you write code, you should regularly add comments to clarify the function of parts of your code. Good commenting can help communicate the intent of your code—both for others *and* for your future self.

Instructions

Try creating one of each type of comment.

Challenge Seed

Solution

```
// Fake Comment
/* Another Comment */
```

2. Declare JavaScript Variables

Description

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`. For example, computers distinguish between numbers, such as the number `12`, and `strings`, such as `"12"`, `"dog"`, or `"123 cats"`, which are collections of characters. Computers can perform mathematical operations on a number, but not on a string. Variables allow computers to store and manipulate data in a dynamic fashion. They do this by using a "label" to point to the data rather than using the data itself. Any of the seven data types may be stored in a variable. `Variables` are similar to the x and y variables you use in mathematics, which means they're a simple name to represent the data we want to refer to. Computer `variables` differ from mathematical variables in that they can store different values at different times. We tell JavaScript to create or declare a variable by putting the keyword `var` in front of it, like so:

```
var ourName;
```

creates a `variable` called `ourName`. In JavaScript we end statements with semicolons. `Variable` names can be made up of numbers, letters, and `$` or `_`, but may not contain spaces or start with a number.

Instructions

Use the `var` keyword to create a variable called `myName`. **Hint**
Look at the `ourName` example if you get stuck.

Challenge Seed

```
// Example
var ourName;

// Declare myName below this line
```

After Test

```
if(typeof myName !== "undefined"){(function(v){return v;})(myName);}
```

Solution

```
var myName;
```

3. Storing Values with the Assignment Operator

Description

In JavaScript, you can store a value in a variable with the assignment operator. `myVariable = 5;` This assigns the `Number` value `5` to `myVariable`. Assignment always goes from right to left. Everything to the right of the `=` operator is resolved before the value is assigned to the variable to the left of the operator.

> myVar = 5;
> myNum = myVar;

This assigns `5` to `myVar` and then resolves `myVar` to `5` again and assigns it to `myNum`.

Instructions

Assign the value `7` to variable `a`. Assign the contents of `a` to variable `b`.

Challenge Seed

```
// Setup
var a;
var b = 2;

// Only change code below this line
```

Before Test

```
if (typeof a != 'undefined') {
  a = undefined;
}
if (typeof b != 'undefined') {
  b = undefined;
}
```

After Test

```
(function(a,b){return "a = " + a + ", b = " + b;})(a,b);
```

Solution

```
var a;
var b = 2;
a = 7;
b = a;
```

4. Initializing Variables with the Assignment Operator

Description

It is common to initialize a variable to an initial value in the same line as it is declared. `var myVar = 0;` Creates a new variable called `myVar` and assigns it an initial value of `0`.

Instructions

Define a variable `a` with `var` and initialize it to a value of `9`.

Challenge Seed

```
// Example
var ourVar = 19;

// Only change code below this line
```

After Test

```
if(typeof a !== 'undefined') {(function(a){return "a = " + a;})(a);} else { (function() {return 'a is
undefined';})(); }
```

Solution

```
var a = 9;
```

5. Understanding Uninitialized Variables

Description

When JavaScript variables are declared, they have an initial value of `undefined`. If you do a mathematical operation on an `undefined` variable your result will be `NaN` which means "Not a Number". If you concatenate a string with an `undefined` variable, you will get a literal string of `"undefined"`.

Instructions

Initialize the three variables `a`, `b`, and `c` with `5`, `10`, and `"I am a"` respectively so that they will not be `undefined`.

Challenge Seed

```
// Initialize these three variables
var a;
var b;
```

```
var c;

// Do not change code below this line

a = a + 1;
b = b + 5;
c = c + " String!";
```

After Test

```
(function(a,b,c){ return "a = " + a + ", b = " + b + ", c = '" + c + "'"; })(a,b,c);
```

Solution

```
var a = 5;
var b = 10;
var c = "I am a";
a = a + 1;
b = b + 5;
c = c + " String!";
```

6. Understanding Case Sensitivity in Variables

Description

In JavaScript all variables and function names are case sensitive. This means that capitalization matters. `MYVAR` is not the same as `MyVar` nor `myvar`. It is possible to have multiple distinct variables with the same name but different casing. It is strongly recommended that for the sake of clarity, you *do not* use this language feature.

Best Practice

Write variable names in JavaScript in camelCase. In camelCase, multi-word variable names have the first word in lowercase and the first letter of each subsequent word is capitalized. **Examples:**

> var someVariable;
> var anotherVariableName;
> var thisVariableNameIsSoLong;

Instructions

Modify the existing declarations and assignments so their names use camelCase.
Do not create any new variables.

Challenge Seed

```
// Declarations
var StUdLyCapVaR;
var properCamelCase;
var TitleCaseOver;

// Assignments
STUDLYCAPVAR = 10;
PRoperCAmelCAse = "A String";
tITLEcASEoVER = 9000;
```

Solution

```
var studlyCapVar;
var properCamelCase;
```

```
var titleCaseOver;

studlyCapVar = 10;
properCamelCase = "A String";
titleCaseOver = 9000;
```

7. Add Two Numbers with JavaScript

Description

`Number` is a data type in JavaScript which represents numeric data. Now let's try to add two numbers using JavaScript. JavaScript uses the `+` symbol as an addition operation when placed between two numbers. **Example:**

> myVar = 5 + 10; // assigned 15

Instructions

Change the `0` so that sum will equal `20`.

Challenge Seed

```
var sum = 10 + 0;
```

After Test

```
(function(z){return 'sum = '+z;})(sum);
```

Solution

```
var sum = 10 + 10;
```

8. Subtract One Number from Another with JavaScript

Description

We can also subtract one number from another. JavaScript uses the `-` symbol for subtraction.
Example

> myVar = 12 - 6; // assigned 6

Instructions

Change the `0` so the difference is `12`.

Challenge Seed

```
var difference = 45 - 0;
```

After Test

```
(function(z){return 'difference = '+z;})(difference);
```

Solution

```
var difference = 45 - 33;
```

9. Multiply Two Numbers with JavaScript

Description

We can also multiply one number by another. JavaScript uses the `*` symbol for multiplication of two numbers.
Example

> myVar = 13 * 13; // assigned 169

Instructions

Change the `0` so that product will equal `80`.

Challenge Seed

```
var product = 8 * 0;
```

After Test

```
(function(z){return 'product = '+z;})(product);
```

Solution

```
var product = 8 * 10;
```

10. Divide One Number by Another with JavaScript

Description

We can also divide one number by another. JavaScript uses the `/` symbol for division.
Example

> myVar = 16 / 2; // assigned 8

Instructions

Change the `0` so that the `quotient` is equal to `2`.

Challenge Seed

```
var quotient = 66 / 0;
```

After Test

```
(function(z){return 'quotient = '+z;})(quotient);
```

Solution

```
var quotient = 66 / 33;
```

11. Increment a Number with JavaScript

Description

You can easily increment or add one to a variable with the `++` operator. `i++;` is the equivalent of `i = i + 1;` **Note**
The entire line becomes `i++;`, eliminating the need for the equal sign.

Instructions

Change the code to use the `++` operator on `myVar`. **Hint**
Learn more about Arithmetic operators - Increment (++).

Challenge Seed

```
var myVar = 87;

// Only change code below this line
myVar = myVar + 1;
```

After Test

```
(function(z){return 'myVar = ' + z;})(myVar);
```

Solution

```
var myVar = 87;
myVar++;
```

12. Decrement a Number with JavaScript

Description

You can easily decrement or decrease a variable by one with the `--` operator. `i--;` is the equivalent of `i = i - 1;`
Note
The entire line becomes `i--;`, eliminating the need for the equal sign.

Instructions

Change the code to use the `--` operator on `myVar`.

Challenge Seed

```
var myVar = 11;
```

```
// Only change code below this line
myVar = myVar - 1;
```

After Test

```
(function(z){return 'myVar = ' + z;})(myVar);
```

Solution

```
var myVar = 11;
myVar--;
```

13. Create Decimal Numbers with JavaScript

Description

We can store decimal numbers in variables too. Decimal numbers are sometimes referred to as floating point numbers or floats. **Note**
Not all real numbers can accurately be represented in floating point. This can lead to rounding errors. Details Here.

Instructions

Create a variable `myDecimal` and give it a decimal value with a fractional part (e.g. `5.7`).

Challenge Seed

```
var ourDecimal = 5.7;

// Only change code below this line
```

After Test

```
(function(){if(typeof myDecimal !== "undefined"){return myDecimal;}})();
```

Solution

```
var myDecimal = 9.9;
```

14. Multiply Two Decimals with JavaScript

Description

In JavaScript, you can also perform calculations with decimal numbers, just like whole numbers. Let's multiply two decimals together to get their product.

Instructions

Change the `0.0` so that product will equal `5.0`.

Challenge Seed

```
var product = 2.0 * 0.0;
```

After Test

```
(function(y){return 'product = '+y;})(product);
```

Solution

```
var product = 2.0 * 2.5;
```

15. Divide One Decimal by Another with JavaScript

Description

Now let's divide one decimal by another.

Instructions

Change the `0.0` so that `quotient` will equal to `2.2`.

Challenge Seed

```
var quotient = 0.0 / 2.0; // Fix this line
```

After Test

```
(function(y){return 'quotient = '+y;})(quotient);
```

Solution

```
var quotient = 4.4 / 2.0;
```

16. Finding a Remainder in JavaScript

Description

The remainder operator `%` gives the remainder of the division of two numbers. **Example**

> 5 % 2 = 1 because
> Math.floor(5 / 2) = 2 (Quotient)
> 2 * 2 = 4
> 5 - 4 = 1 (Remainder)

Usage
In mathematics, a number can be checked to be even or odd by checking the remainder of the division of the number by `2`.

> 17 % 2 = 1 (17 is Odd)
> 48 % 2 = 0 (48 is Even)

Note
The remainder operator is sometimes incorrectly referred to as the "modulus" operator. It is very similar to modulus, but does not work properly with negative numbers.

Instructions

Set `remainder` equal to the remainder of `11` divided by `3` using the remainder (`%`) operator.

Challenge Seed

```
// Only change code below this line

var remainder;
```

After Test

```
(function(y){return 'remainder = '+y;})(remainder);
```

Solution

```
var remainder =  11 % 3;
```

17. Compound Assignment With Augmented Addition

Description

In programming, it is common to use assignments to modify the contents of a variable. Remember that everything to the right of the equals sign is evaluated first, so we can say: `myVar = myVar + 5;` to add `5` to `myVar` . Since this is such a common pattern, there are operators which do both a mathematical operation and assignment in one step. One such operator is the `+=` operator.

> var myVar = 1;
> myVar += 5;
> console.log(myVar); // Returns 6

Instructions

Convert the assignments for `a` , `b` , and `c` to use the `+=` operator.

Challenge Seed

```
var a = 3;
var b = 17;
var c = 12;

// Only modify code below this line

a = a + 12;
b = 9 + b;
c = c + 7;
```

After Test

```
(function(a,b,c){ return "a = " + a + ", b = " + b + ", c = " + c; })(a,b,c);
```

Solution

```
var a = 3;
var b = 17;
var c = 12;

a += 12;
b += 9;
c += 7;
```

18. Compound Assignment With Augmented Subtraction

Description

Like the `+=` operator, `-=` subtracts a number from a variable. `myVar = myVar - 5;` will subtract `5` from `myVar`. This can be rewritten as: `myVar -= 5;`

Instructions

Convert the assignments for `a`, `b`, and `c` to use the `-=` operator.

Challenge Seed

```
var a = 11;
var b = 9;
var c = 3;

// Only modify code below this line

a = a - 6;
b = b - 15;
c = c - 1;
```

After Test

```
(function(a,b,c){ return "a = " + a + ", b = " + b + ", c = " + c; })(a,b,c);
```

Solution

```
var a = 11;
var b = 9;
var c = 3;

a -= 6;
b -= 15;
c -= 1;
```

19. Compound Assignment With Augmented Multiplication

Description

The `*=` operator multiplies a variable by a number. `myVar = myVar * 5;` will multiply `myVar` by `5`. This can be rewritten as: `myVar *= 5;`

Instructions

Convert the assignments for `a`, `b`, and `c` to use the `*=` operator.

Challenge Seed

```
var a = 5;
var b = 12;
var c = 4.6;

// Only modify code below this line

a = a * 5;
b = 3 * b;
c = c * 10;
```

After Test

```
(function(a,b,c){ return "a = " + a + ", b = " + b + ", c = " + c; })(a,b,c);
```

Solution

```
var a = 5;
var b = 12;
var c = 4.6;

a *= 5;
b *= 3;
c *= 10;
```

20. Compound Assignment With Augmented Division

Description

The `/=` operator divides a variable by another number. `myVar = myVar / 5;` Will divide `myVar` by `5`. This can be rewritten as: `myVar /= 5;`

Instructions

Convert the assignments for `a`, `b`, and `c` to use the `/=` operator.

Challenge Seed

```
var a = 48;
var b = 108;
var c = 33;

// Only modify code below this line

a = a / 12;
b = b / 4;
c = c / 11;
```

After Test

```
(function(a,b,c){ return "a = " + a + ", b = " + b + ", c = " + c; })(a,b,c);
```

Solution

```
var a = 48;
var b = 108;
var c = 33;

a /= 12;
b /= 4;
c /= 11;
```

21. Declare String Variables

Description

Previously we have used the code `var myName = "your name";` `"your name"` is called a string literal. It is a string because it is a series of zero or more characters enclosed in single or double quotes.

Instructions

Create two new `string` variables: `myFirstName` and `myLastName` and assign them the values of your first and last name, respectively.

Challenge Seed

```
// Example
var firstName = "Alan";
var lastName = "Turing";

// Only change code below this line
```

After Test

```
if(typeof myFirstName !== "undefined" && typeof myLastName !== "undefined"){(function(){return
myFirstName + ', ' + myLastName;})();}
```

Solution

```
var myFirstName = "Alan";
var myLastName = "Turing";
```

22. Escaping Literal Quotes in Strings

Description

When you are defining a string you must start and end with a single or double quote. What happens when you need a literal quote: `"` or `'` inside of your string? In JavaScript, you can escape a quote from considering it as an end of string quote by placing a backslash (`\`) in front of the quote. `var sampleStr = "Alan said, \"Peter is learning JavaScript\".";` This signals to JavaScript that the following quote is not the end of the string, but should instead

appear inside the string. So if you were to print this to the console, you would get: `Alan said, "Peter is learning JavaScript".`

Instructions

Use backslashes to assign a string to the `myStr` variable so that if you were to print it to the console, you would see: `I am a "double quoted" string inside "double quotes".`

Challenge Seed

```
var myStr = ""; // Change this line
```

After Test

```
(function(){
  if(typeof myStr === 'string') {
    console.log("myStr = \"" + myStr + "\"");
  } else {
    console.log("myStr is undefined");
  }
})();
```

Solution

```
var myStr = "I am a \"double quoted\" string inside \"double quotes\".";
```

23. Quoting Strings with Single Quotes

Description

String values in JavaScript may be written with single or double quotes, as long as you start and end with the same type of quote. Unlike some other programming languages, single and double quotes work the same in JavaScript.

> doubleQuoteStr = "This is a string";
> singleQuoteStr = 'This is also a string';

The reason why you might want to use one type of quote over the other is if you want to use both in a string. This might happen if you want to save a conversation in a string and have the conversation in quotes. Another use for it would be saving an `<a>` tag with various attributes in quotes, all within a string.

> conversation = 'Finn exclaims to Jake, "Algebraic!"';

However, this becomes a problem if you need to use the outermost quotes within it. Remember, a string has the same kind of quote at the beginning and end. But if you have that same quote somewhere in the middle, the string will stop early and throw an error.

> goodStr = 'Jake asks Finn, "Hey, let\'s go on an adventure?"';
> badStr = 'Finn responds, "Let's go!"'; // Throws an error

In the goodStr above, you can use both quotes safely by using the backslash `\` as an escape character. **Note**
The backslash `\` should not be confused with the forward slash `/`. They do not do the same thing.

Instructions

Change the provided string to a string with single quotes at the beginning and end and no escape characters. Right now, the `<a>` tag in the string uses double quotes everywhere. You will need to change the outer quotes to single quotes so you can remove the escape characters.

Challenge Seed

```
var myStr = "<a href=\"http://www.example.com\" target=\"_blank\">Link</a>";
```

After Test

```
(function() { return "myStr = " + myStr; })();
```

Solution

```
var myStr = '<a href="http://www.example.com" target="_blank">Link</a>';
```

24. Escape Sequences in Strings

Description

Quotes are not the only characters that can be escaped inside a string. There are two reasons to use escaping characters:

1. To allow you to use characters you may not otherwise be able to type out, such as a carriage returns.
2. To allow you to represent multiple quotes in a string without JavaScript misinterpreting what you mean.

We learned this in the previous challenge.

Code	Output
`\'`	single quote
`\"`	double quote
`\\`	backslash
`\n`	newline
`\r`	carriage return
`\t`	tab
`\b`	word boundary
`\f`	form feed

Note that the backslash itself must be escaped in order to display as a backslash.

Instructions

Assign the following three lines of text into the single variable `myStr` using escape sequences.

> FirstLine
> \\SecondLine
> ThirdLine

You will need to use escape sequences to insert special characters correctly. You will also need to follow the spacing as it looks above, with no spaces between escape sequences or words. Here is the text with the escape sequences written out. "FirstLine `newline` `tab` `backslash` SecondLine `newline` ThirdLine"

Challenge Seed

```
var myStr; // Change this line
```

After Test

```
(function(){
if (myStr !== undefined){
console.log('myStr:\n' + myStr);}})();
```

Solution

```
var myStr = "FirstLine\n\t\\SecondLine\nThirdLine";
```

25. Concatenating Strings with Plus Operator

Description

In JavaScript, when the `+` operator is used with a `String` value, it is called the concatenation operator. You can build a new string out of other strings by concatenating them together. **Example**

> 'My name is Alan,' + ' I concatenate.'

Note
Watch out for spaces. Concatenation does not add spaces between concatenated strings, so you'll need to add them yourself.

Instructions

Build `myStr` from the strings `"This is the start. "` and `"This is the end."` using the `+` operator.

Challenge Seed

```
// Example
var ourStr = "I come first. " + "I come second.";

// Only change code below this line

var myStr;
```

After Test

```
(function(){
  if(typeof myStr === 'string') {
    return 'myStr = "' + myStr + '"';
  } else {
    return 'myStr is not a string';
  }
})();
```

Solution

```
var ourStr = "I come first. " + "I come second.";
var myStr = "This is the start. " + "This is the end.";
```

26. Concatenating Strings with the Plus Equals Operator

Description

We can also use the `+=` operator to concatenate a string onto the end of an existing string variable. This can be very helpful to break a long string over several lines. **Note**
Watch out for spaces. Concatenation does not add spaces between concatenated strings, so you'll need to add them yourself.

Instructions

Build `myStr` over several lines by concatenating these two strings: `"This is the first sentence. "` and `"This is the second sentence."` using the `+=` operator. Use the `+=` operator similar to how it is shown in the editor. Start by assigning the first string to `myStr`, then add on the second string.

Challenge Seed

```
// Example
var ourStr = "I come first. ";
ourStr += "I come second.";

// Only change code below this line

var myStr;
```

After Test

```
(function(){
  if(typeof myStr === 'string') {
    return 'myStr = "' + myStr + '"';
  } else {
    return 'myStr is not a string';
  }
})();
```

Solution

```
var ourStr = "I come first. ";
ourStr += "I come second.";

var myStr = "This is the first sentence. ";
myStr += "This is the second sentence.";
```

27. Constructing Strings with Variables

Description

Sometimes you will need to build a string, Mad Libs style. By using the concatenation operator (`+`), you can insert one or more variables into a string you're building.

Instructions

Set `myName` to a string equal to your name and build `myStr` with `myName` between the strings `"My name is "` and `" and I am well!"`

Challenge Seed

```
// Example
var ourName = "freeCodeCamp";
var ourStr = "Hello, our name is " + ourName + ", how are you?";

// Only change code below this line
var myName;
var myStr;
```

After Test

```
(function(){
  var output = [];
  if(typeof myName === 'string') {
    output.push('myName = "' + myName + '"');
  } else {
    output.push('myName is not a string');
  }
  if(typeof myStr === 'string') {
    output.push('myStr = "' + myStr + '"');
  } else {
    output.push('myStr is not a string');
  }
  return output.join('\n');
})();
```

Solution

```
var myName = "Bob";
var myStr = "My name is " + myName + " and I am well!";
```

28. Appending Variables to Strings

Description

Just as we can build a string over multiple lines out of string literals, we can also append variables to a string using the plus equals (`+=`) operator.

Instructions

Set `someAdjective` and append it to `myStr` using the `+=` operator.

Challenge Seed

```
// Example
var anAdjective = "awesome!";
var ourStr = "freeCodeCamp is ";
ourStr += anAdjective;

// Only change code below this line

var someAdjective;
var myStr = "Learning to code is ";
```

After Test

```
(function(){
  var output = [];
  if(typeof someAdjective === 'string') {
    output.push('someAdjective = "' + someAdjective + '"');
```

```
  } else {
    output.push('someAdjective is not a string');
  }
  if(typeof myStr === 'string') {
    output.push('myStr = "' + myStr + '"');
  } else {
    output.push('myStr is not a string');
  }
  return output.join('\n');
})();
```

Solution

```
var anAdjective = "awesome!";
var ourStr = "freeCodeCamp is ";
ourStr += anAdjective;

var someAdjective = "neat";
var myStr = "Learning to code is ";
myStr += someAdjective;
```

29. Find the Length of a String

Description

You can find the length of a `String` value by writing `.length` after the string variable or string literal. `"Alan Peter".length; // 10` For example, if we created a variable `var firstName = "Charles"`, we could find out how long the string `"Charles"` is by using the `firstName.length` property.

Instructions

Use the `.length` property to count the number of characters in the `lastName` variable and assign it to `lastNameLength`.

Challenge Seed

```
// Example
var firstNameLength = 0;
var firstName = "Ada";

firstNameLength = firstName.length;

// Setup
var lastNameLength = 0;
var lastName = "Lovelace";

// Only change code below this line.

lastNameLength = lastName;
```

After Test

```
if(typeof lastNameLength !== "undefined"){(function(){return lastNameLength;})();}
```

Solution

```
var firstNameLength = 0;
var firstName = "Ada";
firstNameLength = firstName.length;
```

```
var lastNameLength = 0;
var lastName = "Lovelace";
lastNameLength = lastName.length;
```

30. Use Bracket Notation to Find the First Character in a String

Description

`Bracket notation` is a way to get a character at a specific `index` within a string. Most modern programming languages, like JavaScript, don't start counting at 1 like humans do. They start at 0. This is referred to as Zero-based indexing. For example, the character at index 0 in the word "Charles" is "C". So if `var firstName = "Charles"`, you can get the value of the first letter of the string by using `firstName[0]`.

Instructions

Use bracket notation to find the first character in the `lastName` variable and assign it to `firstLetterOfLastName`. **Hint** Try looking at the `firstLetterOfFirstName` variable declaration if you get stuck.

Challenge Seed

```
// Example
var firstLetterOfFirstName = "";
var firstName = "Ada";

firstLetterOfFirstName = firstName[0];

// Setup
var firstLetterOfLastName = "";
var lastName = "Lovelace";

// Only change code below this line
firstLetterOfLastName = lastName;
```

After Test

```
(function(v){return v;})(firstLetterOfLastName);
```

Solution

```
var firstLetterOfLastName = "";
var lastName = "Lovelace";

// Only change code below this line
firstLetterOfLastName = lastName[0];
```

31. Understand String Immutability

Description

In JavaScript, `String` values are immutable, which means that they cannot be altered once created. For example, the following code:

```
var myStr = "Bob";
myStr[0] = "J";
```

cannot change the value of `myStr` to "Job", because the contents of `myStr` cannot be altered. 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";
```

Instructions

Correct the assignment to `myStr` so it contains the string value of `Hello World` using the approach shown in the example above.

Challenge Seed

```
// Setup
var myStr = "Jello World";

// Only change code below this line

myStr[0] = "H"; // Fix Me
```

After Test

```
(function(v){return "myStr = " + v;})(myStr);
```

Solution

```
var myStr = "Jello World";
myStr = "Hello World";
```

32. Use Bracket Notation to Find the Nth Character in a String

Description

You can also use bracket notation to get the character at other positions within a string. Remember that computers start counting at `0`, so the first character is actually the zeroth character.

Instructions

Let's try to set `thirdLetterOfLastName` to equal the third letter of the `lastName` variable using bracket notation. **Hint** Try looking at the `secondLetterOfFirstName` variable declaration if you get stuck.

Challenge Seed

```
// Example
var firstName = "Ada";
var secondLetterOfFirstName = firstName[1];

// Setup
var lastName = "Lovelace";

// Only change code below this line.
```

```
var thirdLetterOfLastName = lastName;
```

After Test

```
(function(v){return v;})(thirdLetterOfLastName);
```

Solution

```
var lastName = "Lovelace";
var thirdLetterOfLastName = lastName[2];
```

33. Use Bracket Notation to Find the Last Character in a String

Description

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]`.

Instructions

Use bracket notation to find the last character in the `lastName` variable. **Hint**
Try looking at the `lastLetterOfFirstName` variable declaration if you get stuck.

Challenge Seed

```
// Example
var firstName = "Ada";
var lastLetterOfFirstName = firstName[firstName.length - 1];

// Setup
var lastName = "Lovelace";

// Only change code below this line.
var lastLetterOfLastName = lastName;
```

After Test

```
(function(v){return v;})(lastLetterOfLastName);
```

Solution

```
var firstName = "Ada";
var lastLetterOfFirstName = firstName[firstName.length - 1];

var lastName = "Lovelace";
var lastLetterOfLastName = lastName[lastName.length - 1];
```

34. Use Bracket Notation to Find the Nth-to-Last Character in a String

Description

You can use the same principle we just used to retrieve the last character in a string to retrieve the Nth-to-last character. For example, you can get the value of the third-to-last letter of the `var firstName = "Charles"` string by using `firstName[firstName.length - 3]`

Instructions

Use bracket notation to find the second-to-last character in the `lastName` string. **Hint**
Try looking at the `thirdToLastLetterOfFirstName` variable declaration if you get stuck.

Challenge Seed

```
// Example
var firstName = "Ada";
var thirdToLastLetterOfFirstName = firstName[firstName.length - 3];

// Setup
var lastName = "Lovelace";

// Only change code below this line
var secondToLastLetterOfLastName = lastName;
```

After Test

```
(function(v){return v;})(secondToLastLetterOfLastName);
```

Solution

```
var firstName = "Ada";
var thirdToLastLetterOfFirstName = firstName[firstName.length - 3];

var lastName = "Lovelace";
var secondToLastLetterOfLastName = lastName[lastName.length - 2];
```

35. Word Blanks

Description

We will now use our knowledge of strings to build a "Mad Libs" style word game we're calling "Word Blanks". You will create an (optionally humorous) "Fill in the Blanks" style sentence. In a "Mad Libs" game, you are provided sentences with some missing words, like nouns, verbs, adjectives and adverbs. You then fill in the missing pieces with words of your choice in a way that the completed sentence makes sense. Consider this sentence - "It was really ___, and we ___ ourselves ___". This sentence has three missing pieces- an adjective, a verb and an adverb, and we can add words of our choice to complete it. We can then assign the completed sentence to a variable as follows:

> var sentence = "It was really" + "hot" + ", and we" + "laughed" + "ourselves" + "silly.";

Instructions

In this challenge, we provide you with a noun, a verb, an adjective and an adverb. You need to form a complete sentence using words of your choice, along with the words we provide. You will need to use the string concatenation operator `+` to build a new string, using the provided variables: `myNoun`, `myAdjective`, `myVerb`, and `myAdverb`. You will then assign the formed string to the `result` variable. You will also need to account for spaces in your string, so that the final sentence has spaces between all the words. The result should be a complete sentence.

Challenge Seed

```
function wordBlanks(myNoun, myAdjective, myVerb, myAdverb) {
  // Your code below this line
  var result = "";

  // Your code above this line
  return result;
}

// Change the words here to test your function
wordBlanks("dog", "big", "ran", "quickly");
```

After Test

```
var test1 = wordBlanks("dog", "big", "ran", "quickly");
var test2 = wordBlanks("cat", "little", "hit", "slowly");
```

Solution

```
function wordBlanks(myNoun, myAdjective, myVerb, myAdverb) {
  var result = "";

  result = "Once there was a " + myNoun + " which was very " + myAdjective + ". ";
  result += "It " + myVerb + " " + myAdverb + " around the yard.";

  return result;
}
```

36. Store Multiple Values in one Variable using JavaScript Arrays

Description

With JavaScript `array` variables, we can store several pieces of data in one place. You start an array declaration with an opening square bracket, end it with a closing square bracket, and put a comma between each entry, like this: `var sandwich = ["peanut butter", "jelly", "bread"]`.

Instructions

Modify the new array `myArray` so that it contains both a `string` and a `number` (in that order). **Hint**
Refer to the example code in the text editor if you get stuck.

Challenge Seed

```
// Example
var ourArray = ["John", 23];

// Only change code below this line.
var myArray = [];
```

After Test

```
(function(z){return z;})(myArray);
```

Solution

```
var myArray = ["The Answer", 42];
```

37. Nest one Array within Another Array

Description

You can also nest arrays within other arrays, like this: `[["Bulls", 23], ["White Sox", 45]]`. This is also called a Multi-dimensional Array.

Instructions

Create a nested array called `myArray`.

Challenge Seed

```
// Example
var ourArray = [["the universe", 42], ["everything", 101010]];

// Only change code below this line.
var myArray = [];
```

After Test

```
if(typeof myArray !== "undefined"){(function(){return myArray;})();}
```

Solution

```
var myArray = [[1,2,3]];
```

38. Access Array Data with Indexes

Description

We can access the data inside arrays 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. Like strings, arrays use zero-based indexing, so the first element in an array is element `0`.

Example

> var array = [50,60,70];
> array[0]; // equals 50
> var data = array[1]; // equals 60

Note

There shouldn't be any spaces between the array name and the square brackets, like `array [0]`. Although JavaScript is able to process this correctly, this may confuse other programmers reading your code.

Instructions

Create a variable called `myData` and set it to equal the first value of `myArray` using bracket notation.

Challenge Seed

```
// Example
var ourArray = [50,60,70];
var ourData = ourArray[0]; // equals 50

// Setup
var myArray = [50,60,70];

// Only change code below this line.
```

After Test

```
if(typeof myArray !== "undefined" && typeof myData !== "undefined"){(function(y,z){return 'myArray = ' +
JSON.stringify(y) + ', myData = ' + JSON.stringify(z);})(myArray, myData);}
```

Solution

```
var myArray = [50,60,70];
var myData = myArray[0];
```

39. Modify Array Data With Indexes

Description

Unlike strings, the entries of arrays are mutable and can be changed freely. **Example**

> var ourArray = [50,40,30];
> ourArray[0] = 15; // equals [15,40,30]

Note
There shouldn't be any spaces between the array name and the square brackets, like `array [0]`. Although JavaScript is able to process this correctly, this may confuse other programmers reading your code.

Instructions

Modify the data stored at index `0` of `myArray` to a value of `45`.

Challenge Seed

```
// Example
var ourArray = [18,64,99];
ourArray[1] = 45; // ourArray now equals [18,45,99].

// Setup
var myArray = [18,64,99];

// Only change code below this line.
```

After Test

```
if(typeof myArray !== "undefined"){(function(){return myArray;})();}
```

Solution

```
var myArray = [18,64,99];
myArray[0] = 45;
```

40. Access Multi-Dimensional Arrays With Indexes

Description

One way to think of a multi-dimensional array, is as an *array of arrays*. When you use brackets to access your array, the first set of brackets refers to the entries in the outer-most (the first level) array, and each additional pair of brackets refers to the next level of entries inside. **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
```

Note
There shouldn't be any spaces between the array name and the square brackets, like `array [0][0]` and even this `array [0] [0]` is not allowed. Although JavaScript is able to process this correctly, this may confuse other programmers reading your code.

Instructions

Using bracket notation select an element from `myArray` such that `myData` is equal to `8`.

Challenge Seed

```
// Setup
var myArray = [[1,2,3], [4,5,6], [7,8,9], [[10,11,12], 13, 14]];

// Only change code below this line.
var myData = myArray[0][0];
```

After Test

```
if(typeof myArray !== "undefined"){(function(){return "myData: " + myData + " myArray: " +
JSON.stringify(myArray);})();}
```

Solution

```
var myArray = [[1,2,3],[4,5,6], [7,8,9], [[10,11,12], 13, 14]];
var myData = myArray[2][1];
```

41. Manipulate Arrays With push()

Description

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]
```

Instructions

Push `["dog", 3]` onto the end of the `myArray` variable.

Challenge Seed

```
// Example
var ourArray = ["Stimpson", "J", "cat"];
ourArray.push(["happy", "joy"]);
// ourArray now equals ["Stimpson", "J", "cat", ["happy", "joy"]]

// Setup
var myArray = [["John", 23], ["cat", 2]];

// Only change code below this line.
```

After Test

```
(function(z){return 'myArray = ' + JSON.stringify(z);})(myArray);
```

Solution

```
var myArray = [["John", 23], ["cat", 2]];
myArray.push(["dog",3]);
```

42. Manipulate Arrays With pop()

Description

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. Any type of entry can be "popped" off of an array - numbers, strings, even nested arrays.

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

Instructions

Use the `.pop()` function to remove the last item from `myArray`, assigning the "popped off" value to `removedFromMyArray`.

Challenge Seed

```
// Example
var ourArray = [1,2,3];
var removedFromOurArray = ourArray.pop();
// removedFromOurArray now equals 3, and ourArray now equals [1,2]

// Setup
var myArray = [["John", 23], ["cat", 2]];

// Only change code below this line.
var removedFromMyArray;
```

After Test

```
(function(y, z){return 'myArray = ' + JSON.stringify(y) + ' & removedFromMyArray = ' +
JSON.stringify(z);})(myArray, removedFromMyArray);
```

Solution

```
var myArray = [["John", 23], ["cat", 2]];
var removedFromMyArray = myArray.pop();
```

43. Manipulate Arrays With shift()

Description

`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.

Instructions

Use the `.shift()` function to remove the first item from `myArray`, assigning the "shifted off" value to `removedFromMyArray`.

Challenge Seed

```
// Example
var ourArray = ["Stimpson", "J", ["cat"]];
var removedFromOurArray = ourArray.shift();
// removedFromOurArray now equals "Stimpson" and ourArray now equals ["J", ["cat"]].

// Setup
var myArray = [["John", 23], ["dog", 3]];

// Only change code below this line.
var removedFromMyArray;
```

After Test

```
(function(y, z){return 'myArray = ' + JSON.stringify(y) + ' & removedFromMyArray = ' +
JSON.stringify(z);})(myArray, removedFromMyArray);
```

Solution

```
var myArray = [["John", 23], ["dog", 3]];

// Only change code below this line.
var removedFromMyArray = myArray.shift();
```

44. Manipulate Arrays With unshift()

Description

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.

Instructions

Add `["Paul",35]` to the beginning of the `myArray` variable using `unshift()`.

Challenge Seed

```
// Example
var ourArray = ["Stimpson", "J", "cat"];
ourArray.shift(); // ourArray now equals ["J", "cat"]
ourArray.unshift("Happy");
// ourArray now equals ["Happy", "J", "cat"]

// Setup
var myArray = [["John", 23], ["dog", 3]];
myArray.shift();

// Only change code below this line.
```

After Test

```
(function(y, z){return 'myArray = ' + JSON.stringify(y);})(myArray);
```

Solution

```
var myArray = [["John", 23], ["dog", 3]];
myArray.shift();
myArray.unshift(["Paul", 35]);
```

45. Shopping List

Description

Create a shopping list in the variable `myList`. The list should be a multi-dimensional array containing several sub-arrays. The first element in each sub-array should contain a string with the name of the item. The second element should be a number representing the quantity i.e. `["Chocolate Bar", 15]` There should be at least 5 sub-arrays in the list.

Instructions

Challenge Seed

```
var myList = [];
```

After Test

```
var count = 0;
var isArray = false;
var hasString = false;
var hasNumber = false;
(function(list){
  if(Array.isArray(myList)) {
    isArray = true;
    if(myList.length > 0) {
      hasString = true;
      hasNumber = true;
```

```
      for (var elem of myList) {
        if(!elem || !elem[0] || typeof elem[0] !== 'string') {
          hasString = false;
        }
        if(!elem || typeof elem[1] !== 'number') {
          hasNumber = false;
        }
      }
    }
    count = myList.length;
    return JSON.stringify(myList);
  } else {
    return "myList is not an array";
  }

})(myList);
```

Solution

```
var myList = [
  ["Candy", 10],
  ["Potatoes", 12],
  ["Eggs", 12],
  ["Catfood", 1],
  ["Toads", 9]
];
```

46. Write Reusable JavaScript with Functions

Description

In JavaScript, we can divide up our code into reusable parts called functions. Here's an example of a function:

```
function functionName() {
  console.log("Hello World");
}
```

You can call or invoke this function by using its name followed by parentheses, like this: `functionName();` Each time the function is called it will print out the message `"Hello World"` on the dev console. All of the code between the curly braces will be executed every time the function is called.

Instructions

1. Create a function called `reusableFunction` which prints `"Hi World"` to the dev console.
2. Call the function.

Challenge Seed

```
// Example
function ourReusableFunction() {
  console.log("Heyya, World");
}

ourReusableFunction();

// Only change code below this line
```

Before Test

```
var logOutput = "";
var originalConsole = console
function capture() {
```

```
    var nativeLog = console.log;
    console.log = function (message) {
        if(message && message.trim) logOutput = message.trim();
        if(nativeLog.apply) {
          nativeLog.apply(originalConsole, arguments);
        } else {
          var nativeMsg = Array.prototype.slice.apply(arguments).join(' ');
          nativeLog(nativeMsg);
        }
    };
}

function uncapture() {
  console.log = originalConsole.log;
}

capture();
```

After Test

```
uncapture();

if (typeof reusableFunction !== "function") {
  (function() { return "reusableFunction is not defined"; })();
} else {
  (function() { return logOutput || "console.log never called"; })();
}
```

Solution

```
function reusableFunction() {
  console.log("Hi World");
}
reusableFunction();
```

47. Passing Values to Functions with Arguments

Description

Parameters are variables that act as placeholders for the values that are to be input to a function when it is called. When a function is defined, it is typically defined along with one or more parameters. The actual values that are input (or "passed") into a function when it is called are known as arguments. Here is a function with two parameters, `param1` and `param2`:

> function testFun(param1, param2) {
> console.log(param1, param2);
> }

Then we can call `testFun`: `testFun("Hello", "World");` We have passed two arguments, `"Hello"` and `"World"`. Inside the function, `param1` will equal "Hello" and `param2` will equal "World". Note that you could call `testFun` again with different arguments and the parameters would take on the value of the new arguments.

Instructions

1. Create a function called `functionWithArgs` that accepts two arguments and outputs their sum to the dev console.
2. Call the function with two numbers as arguments.

Challenge Seed

```
// Example
function ourFunctionWithArgs(a, b) {
  console.log(a - b);
```

```
}
ourFunctionWithArgs(10, 5); // Outputs 5

// Only change code below this line.
```

Before Test

```
var logOutput = "";
var originalConsole = console
function capture() {
    var nativeLog = console.log;
    console.log = function (message) {
        if(message) logOutput = JSON.stringify(message).trim();
        if(nativeLog.apply) {
          nativeLog.apply(originalConsole, arguments);
        } else {
          var nativeMsg = Array.prototype.slice.apply(arguments).join(' ');
          nativeLog(nativeMsg);
        }
    };
}

function uncapture() {
  console.log = originalConsole.log;
}

capture();
```

After Test

```
uncapture();

if (typeof functionWithArgs !== "function") {
  (function() { return "functionWithArgs is not defined"; })();
} else {
  (function() { return logOutput || "console.log never called"; })();
}
```

Solution

```
function functionWithArgs(a, b) {
  console.log(a + b);
}
functionWithArgs(10, 5);
```

48. Global Scope and Functions

Description

In JavaScript, scope refers to the visibility of variables. Variables which are defined outside of a function block have Global scope. This means, they can be seen everywhere in your JavaScript code. 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`.

Instructions

Using `var`, declare a `global` variable `myGlobal` outside of any function. Initialize it with a value of `10`. Inside function `fun1`, assign `5` to `oopsGlobal` ***without*** using the `var` keyword.

Challenge Seed

```
// Declare your variable here


function fun1() {
  // Assign 5 to oopsGlobal Here

}

// Only change code above this line
function fun2() {
  var output = "";
  if (typeof myGlobal != "undefined") {
    output += "myGlobal: " + myGlobal;
  }
  if (typeof oopsGlobal != "undefined") {
    output += " oopsGlobal: " + oopsGlobal;
  }
  console.log(output);
}
```

Before Test

```
var logOutput = "";
var originalConsole = console
function capture() {
    var nativeLog = console.log;
    console.log = function (message) {
        logOutput = message;
        if(nativeLog.apply) {
          nativeLog.apply(originalConsole, arguments);
        } else {
          var nativeMsg = Array.prototype.slice.apply(arguments).join(' ');
          nativeLog(nativeMsg);
        }
    };
}

function uncapture() {
  console.log = originalConsole.log;
}
var oopsGlobal;
capture();
```

After Test

```
fun1();
fun2();
uncapture();
(function() { return logOutput || "console.log never called"; })();
```

Solution

```
// Declare your variable here
var myGlobal = 10;

function fun1() {
  // Assign 5 to oopsGlobal Here
  oopsGlobal = 5;
}

// Only change code above this line
function fun2() {
  var output = "";
  if(typeof myGlobal != "undefined") {
    output += "myGlobal: " + myGlobal;
  }
  if(typeof oopsGlobal != "undefined") {
    output += " oopsGlobal: " + oopsGlobal;
  }
```

```
  console.log(output);
}
```

49. Local Scope and Functions

Description

Variables which are declared within a function, as well as the function parameters have local scope. That means, they are only visible within that function. Here is a function `myTest` with a local variable called `loc`.

```
function myTest() {
  var loc = "foo";
  console.log(loc);
}
myTest(); // logs "foo"
console.log(loc); // loc is not defined
```

`loc` is not defined outside of the function.

Instructions

Declare a local variable `myVar` inside `myLocalScope`. Run the tests and then follow the instructions commented out in the editor. **Hint**
Refreshing the page may help if you get stuck.

Challenge Seed

```
function myLocalScope() {
  'use strict'; // you shouldn't need to edit this line

  console.log(myVar);
}
myLocalScope();

// Run and check the console
// myVar is not defined outside of myLocalScope
console.log(myVar);

// Now remove the console log line to pass the test
```

Before Test

```
var logOutput = "";
var originalConsole = console
function capture() {
  var nativeLog = console.log;
  console.log = function (message) {
    logOutput = message;
    if(nativeLog.apply) {
      nativeLog.apply(originalConsole, arguments);
    } else {
      var nativeMsg = Array.prototype.slice.apply(arguments).join(' ');
      nativeLog(nativeMsg);
    }
  };
}

function uncapture() {
  console.log = originalConsole.log;
}
```

After Test

```
typeof myLocalScope === 'function' && (capture(), myLocalScope(), uncapture());
(function() { return logOutput || "console.log never called"; })();
```

Solution

```
function myLocalScope() {
  'use strict';

  var myVar;
  console.log(myVar);
}
myLocalScope();
```

50. Global vs. Local Scope in Functions

Description

It is possible to have both local and global variables with the same name. When you do this, the `local` variable takes precedence over the `global` variable. In this example:

```
var someVar = "Hat";
function myFun() {
  var someVar = "Head";
  return someVar;
}
```

The function `myFun` will return `"Head"` because the `local` version of the variable is present.

Instructions

Add a local variable to `myOutfit` function to override the value of `outerWear` with `"sweater"`.

Challenge Seed

```
// Setup
var outerWear = "T-Shirt";

function myOutfit() {
  // Only change code below this line



  // Only change code above this line
  return outerWear;
}

myOutfit();
```

Solution

```
var outerWear = "T-Shirt";
function myOutfit() {
  var outerWear = "sweater";
  return outerWear;
}
```

51. Return a Value from a Function with Return

Description

We can pass values into a function with arguments. You can use a `return` statement to send a value back out of a function. **Example**

```
function plusThree(num) {
  return num + 3;
}
var answer = plusThree(5); // 8
```

`plusThree` takes an argument for `num` and returns a value equal to `num + 3`.

Instructions

Create a function `timesFive` that accepts one argument, multiplies it by `5`, and returns the new value. See the last line in the editor for an example of how you can test your `timesFive` function.

Challenge Seed

```
// Example
function minusSeven(num) {
  return num - 7;
}

// Only change code below this line


console.log(minusSeven(10));
```

Solution

```
function timesFive(num) {
  return num * 5;
}
timesFive(10);
```

52. Understanding Undefined Value returned from a Function

Description

A function can include the `return` statement but it does not have to. In the case that the function doesn't have a `return` statement, when you call it, the function processes the inner code but the returned value is `undefined`. **Example**

```
var sum = 0;
function addSum(num) {
  sum = sum + num;
}
var returnedValue = addSum(3); // sum will be modified but returned value is undefined
```

`addSum` is a function without a `return` statement. The function will change the global `sum` variable but the returned value of the function is `undefined`

Instructions

Create a function `addFive` without any arguments. This function adds 5 to the `sum` variable, but its returned value is `undefined`.

Challenge Seed

```
// Example
var sum = 0;
function addThree() {
  sum = sum + 3;
}

// Only change code below this line



// Only change code above this line
var returnedValue = addFive();
```

After Test

```
var sum = 0;
function addThree() {sum = sum + 3;}
addThree();
addFive();
```

Solution

```
function addFive() {
 sum = sum + 5;
}
```

53. Assignment with a Returned Value

Description

If you'll recall from our discussion of Storing Values with the Assignment Operator, everything to the right of the equal sign is resolved before the value is assigned. This means we can take the return value of a function and assign it to a variable. Assume we have pre-defined a function `sum` which adds two numbers together, then: `ourSum = sum(5, 12);` will call `sum` function, which returns a value of `17` and assigns it to `ourSum` variable.

Instructions

Call the `processArg` function with an argument of `7` and assign its return value to the variable `processed`.

Challenge Seed

```
// Example
var changed = 0;

function change(num) {
  return (num + 5) / 3;
}

changed = change(10);

// Setup
var processed = 0;

function processArg(num) {
  return (num + 3) / 5;
}

// Only change code below this line
```

After Test

```
(function(){return "processed = " + processed})();
```

Solution

```
var processed = 0;

function processArg(num) {
  return (num + 3) / 5;
}

processed = processArg(7);
```

54. Stand in Line

Description

In Computer Science a queue is an abstract Data Structure where items are kept in order. New items can be added at the back of the `queue` and old items are taken off from the front of the `queue`. Write a function `nextInLine` which takes an array (`arr`) and a number (`item`) as arguments. Add the number to the end of the array, then remove the first element of the array. The `nextInLine` function should then return the element that was removed.

Instructions

Challenge Seed

```
function nextInLine(arr, item) {
  // Your code here

  return item;  // Change this line
}

// Test Setup
var testArr = [1,2,3,4,5];

// Display Code
console.log("Before: " + JSON.stringify(testArr));
console.log(nextInLine(testArr, 6)); // Modify this line to test
console.log("After: " + JSON.stringify(testArr));
```

Before Test

```
var logOutput = [];
var originalConsole = console
function capture() {
    var nativeLog = console.log;
    console.log = function (message) {
        logOutput.push(message);
        if(nativeLog.apply) {
          nativeLog.apply(originalConsole, arguments);
        } else {
          var nativeMsg = Array.prototype.slice.apply(arguments).join(' ');
          nativeLog(nativeMsg);
        }
    };
}

function uncapture() {
  console.log = originalConsole.log;
```

```
}

capture();
```

After Test

```
uncapture();
testArr = [1,2,3,4,5];
(function() { return logOutput.join("\n");})();
```

Solution

```
var testArr = [ 1,2,3,4,5];

function nextInLine(arr, item) {
    arr.push(item);
    return arr.shift();
}
```

55. Understanding Boolean Values

Description

Another data type is the Boolean. `Booleans` may only be one of two values: `true` or `false`. They are basically little on-off switches, where `true` is "on" and `false` is "off." These two states are mutually exclusive. **Note** `Boolean` values are never written with quotes. The `strings` `"true"` and `"false"` are not `Boolean` and have no special meaning in JavaScript.

Instructions

Modify the `welcomeToBooleans` function so that it returns `true` instead of `false` when the run button is clicked.

Challenge Seed

```
function welcomeToBooleans() {

  // Only change code below this line.

  return false; // Change this line

  // Only change code above this line.
}
```

After Test

```
welcomeToBooleans();
```

Solution

```
function welcomeToBooleans() {
  return true; // Change this line
}
```

56. Use Conditional Logic with If Statements

Description

`If` statements are used to make decisions in code. The keyword `if` tells JavaScript to execute the code in the curly braces under certain conditions, defined in the parentheses. These conditions are known as `Boolean` conditions and they may only be `true` or `false`. When the condition evaluates to `true`, the program executes the statement inside the curly braces. When the Boolean condition evaluates to `false`, the statement inside the curly braces will not execute. **Pseudocode**

> if (*condition is true*) {
> *statement is executed*
> }

Example

```
function test (myCondition) {
  if (myCondition) {
    return "It was true";
  }
  return "It was false";
}
test(true); // returns "It was true"
test(false); // returns "It was false"
```

When `test` is called with a value of `true`, the `if` statement evaluates `myCondition` to see if it is `true` or not. Since it is `true`, the function returns `"It was true"`. When we call `test` with a value of `false`, `myCondition` is *not* `true` and the statement in the curly braces is not executed and the function returns `"It was false"`.

Instructions

Create an `if` statement inside the function to return `"Yes, that was true"` if the parameter `wasThatTrue` is `true` and return `"No, that was false"` otherwise.

Challenge Seed

```
// Example
function ourTrueOrFalse(isItTrue) {
  if (isItTrue) {
    return "Yes, it's true";
  }
  return "No, it's false";
}

// Setup
function trueOrFalse(wasThatTrue) {

  // Only change code below this line.



  // Only change code above this line.

}

// Change this value to test
trueOrFalse(true);
```

Solution

```
function trueOrFalse(wasThatTrue) {
  if (wasThatTrue) {
    return "Yes, that was true";
  }
  return "No, that was false";
}
```

57. Comparison with the Equality Operator

Description

There are many Comparison Operators in JavaScript. All of these operators return a boolean `true` or `false` value. The most basic operator is the equality operator `==`. The equality operator compares two values and returns `true` if they're equivalent or `false` if they are not. Note that equality is different from assignment (`=`), which assigns the value at the right of the operator to a variable in the left.

```
function equalityTest(myVal) {
 if (myVal == 10) {
   return "Equal";
 }
 return "Not Equal";
}
```

If `myVal` is equal to `10`, the equality operator returns `true`, so the code in the curly braces will execute, and the function will return `"Equal"`. Otherwise, the function will return `"Not Equal"`. In order for JavaScript to compare two different `data types` (for example, `numbers` and `strings`), it must convert one type to another. This is known as "Type Coercion". Once it does, however, it can compare terms as follows:

```
1 == 1 // true
1 == 2 // false
1 == '1' // true
"3" == 3 // true
```

Instructions

Add the `equality operator` to the indicated line so that the function will return "Equal" when `val` is equivalent to `12`

Challenge Seed

```
// Setup
function testEqual(val) {
  if (val) { // Change this line
    return "Equal";
  }
  return "Not Equal";
}

// Change this value to test
testEqual(10);
```

Solution

```
function testEqual(val) {
  if (val == 12) {
    return "Equal";
  }
  return "Not Equal";
}
```

58. Comparison with the Strict Equality Operator

Description

Strict equality (`===`) is the counterpart to the equality operator (`==`). However, unlike the equality operator, which attempts to convert both values being compared to a common type, the strict equality operator does not perform a type conversion. If the values being compared have different types, they are considered unequal, and the strict equality operator will return false. **Examples**

> 3 === 3 // true
> 3 === '3' // false

In the second example, `3` is a `Number` type and `'3'` is a `String` type.

Instructions

Use the strict equality operator in the `if` statement so the function will return "Equal" when `val` is strictly equal to `7`

Challenge Seed

```
// Setup
function testStrict(val) {
  if (val) { // Change this line
    return "Equal";
  }
  return "Not Equal";
}

// Change this value to test
testStrict(10);
```

Solution

```
function testStrict(val) {
  if (val === 7) {
    return "Equal";
  }
  return "Not Equal";
}
```

59. Practice comparing different values

Description

In the last two challenges, we learned about the equality operator (`==`) and the strict equality operator (`===`). Let's do a quick review and practice using these operators some more. If the values being compared are not of the same type, the equality operator will perform a type conversion, and then evaluate the values. However, the strict equality operator will compare both the data type and value as-is, without converting one type to the other. **Examples**

> 3 == '3' // returns true because JavaScript performs type conversion from string to number
> 3 === '3' // returns false because the types are different and type conversion is not performed

Note
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'

Instructions

The `compareEquality` function in the editor compares two values using the `equality operator` . Modify the function so that it returns "Equal" only when the values are strictly equal.

Challenge Seed

```
// Setup
function compareEquality(a, b) {
  if (a == b) { // Change this line
    return "Equal";
  }
  return "Not Equal";
}

// Change this value to test
compareEquality(10, "10");
```

Solution

```
function compareEquality(a,b) {
  if (a === b) {
    return "Equal";
  }
  return "Not Equal";
}
```

60. Comparison with the Inequality Operator

Description

The inequality operator (`!=`) is the opposite of the equality operator. It means "Not Equal" and returns `false` where equality would return `true` and *vice versa*. Like the equality operator, the inequality operator will convert data types of values while comparing. **Examples**

> 1 != 2 // true
> 1 != "1" // false
> 1 != '1' // false
> 1 != true // false
> 0 != false // false

Instructions

Add the inequality operator `!=` in the `if` statement so that the function will return "Not Equal" when `val` is not equivalent to `99`

Challenge Seed

```
// Setup
function testNotEqual(val) {
  if (val) { // Change this line
    return "Not Equal";
  }
  return "Equal";
}

// Change this value to test
testNotEqual(10);
```

Solution

```
function testNotEqual(val) {
  if (val != 99) {
    return "Not Equal";
  }
  return "Equal";
}
```

61. Comparison with the Strict Inequality Operator

Description

The strict inequality operator (`!==`) is the logical opposite of the strict equality operator. It means "Strictly Not Equal" and returns `false` where strict equality would return `true` and *vice versa*. Strict inequality will not convert data types.
Examples

> 3 !== 3 // false
> 3 !== '3' // true
> 4 !== 3 // true

Instructions

Add the `strict inequality operator` to the `if` statement so the function will return "Not Equal" when `val` is not strictly equal to `17`

Challenge Seed

```
// Setup
function testStrictNotEqual(val) {
  // Only Change Code Below this Line

  if (val) {

  // Only Change Code Above this Line

    return "Not Equal";
  }
  return "Equal";
}

// Change this value to test
testStrictNotEqual(10);
```

Solution

```
function testStrictNotEqual(val) {
  if (val !== 17) {
    return "Not Equal";
  }
  return "Equal";
}
```

62. Comparison with the Greater Than Operator

Description

The greater than operator (`>`) compares the values of two numbers. If the number to the left is greater than the number to the right, it returns `true` . Otherwise, it returns `false` . Like the equality operator, greater than operator will convert data types of values while comparing. **Examples**

> 5 > 3 // true
> 7 > '3' // true
> 2 > 3 // false
> '1' > 9 // false

Instructions

Add the `greater than` operator to the indicated lines so that the return statements make sense.

Challenge Seed

```
function testGreaterThan(val) {
  if (val) {  // Change this line
    return "Over 100";
  }

  if (val) {  // Change this line
    return "Over 10";
  }

  return "10 or Under";
}

// Change this value to test
testGreaterThan(10);
```

Solution

```
function testGreaterThan(val) {
  if (val > 100) {  // Change this line
    return "Over 100";
  }
  if (val > 10) {  // Change this line
    return "Over 10";
  }
  return "10 or Under";
}
```

63. Comparison with the Greater Than Or Equal To Operator

Description

The `greater than or equal to` operator (`>=`) compares the values of two numbers. If the number to the left is greater than or equal to the number to the right, it returns `true` . Otherwise, it returns `false` . Like the equality operator, `greater than or equal to` operator will convert data types while comparing. **Examples**

> 6 >= 6 // true
> 7 >= '3' // true
> 2 >= 3 // false
> '7' >= 9 // false

Instructions

Add the `greater than or equal to` operator to the indicated lines so that the return statements make sense.

Challenge Seed

```
function testGreaterOrEqual(val) {
  if (val) {  // Change this line
    return "20 or Over";
  }

  if (val) {  // Change this line
    return "10 or Over";
  }

  return "Less than 10";
}
```

```
// Change this value to test
testGreaterOrEqual(10);
```

Solution

```
function testGreaterOrEqual(val) {
  if (val >= 20) {  // Change this line
    return "20 or Over";
  }

  if (val >= 10) {  // Change this line
    return "10 or Over";
  }

  return "Less than 10";
}
```

64. Comparison with the Less Than Operator

Description

The less than operator (`<`) compares the values of two numbers. If the number to the left is less than the number to the right, it returns `true` . Otherwise, it returns `false` . Like the equality operator, less than operator converts data types while comparing. **Examples**

> 2 < 5 // true
> '3' < 7 // true
> 5 < 5 // false
> 3 < 2 // false
> '8' < 4 // false

Instructions

Add the `less than` operator to the indicated lines so that the return statements make sense.

Challenge Seed

```
function testLessThan(val) {
  if (val) {  // Change this line
    return "Under 25";
  }

  if (val) {  // Change this line
    return "Under 55";
  }

  return "55 or Over";
}

// Change this value to test
testLessThan(10);
```

Solution

```
function testLessThan(val) {
  if (val < 25) {  // Change this line
    return "Under 25";
  }

  if (val < 55) {  // Change this line
    return "Under 55";
```

```
  }

  return "55 or Over";
}
```

65. Comparison with the Less Than Or Equal To Operator

Description

The `less than or equal to` operator (`<=`) compares the values of two numbers. If the number to the left is less than or equal to the number to the right, it returns `true` . If the number on the left is greater than the number on the right, it returns `false` . Like the equality operator, `less than or equal to` converts data types. **Examples**

> 4 <= 5 // true
> '7' <= 7 // true
> 5 <= 5 // true
> 3 <= 2 // false
> '8' <= 4 // false

Instructions

Add the `less than or equal to` operator to the indicated lines so that the return statements make sense.

Challenge Seed

```
function testLessOrEqual(val) {
  if (val) {  // Change this line
    return "Smaller Than or Equal to 12";
  }

  if (val) {  // Change this line
    return "Smaller Than or Equal to 24";
  }

  return "More Than 24";
}

// Change this value to test
testLessOrEqual(10);
```

Solution

```
function testLessOrEqual(val) {
  if (val <= 12) {  // Change this line
    return "Smaller Than or Equal to 12";
  }

  if (val <= 24) {  // Change this line
    return "Smaller Than or Equal to 24";
  }

  return "More Than 24";
}
```

66. Comparisons with the Logical And Operator

Description

Sometimes you will need to test more than one thing at a time. The logical and operator (`&&`) returns `true` if and only if the operands to the left and right of it are true. The same effect could be achieved by nesting an if statement inside another if:

```
if (num > 5) {
  if (num < 10) {
    return "Yes";
  }
}
return "No";
```

will only return "Yes" if `num` is greater than `5` and less than `10` . The same logic can be written as:

```
if (num > 5 && num < 10) {
  return "Yes";
}
return "No";
```

Instructions

Combine the two if statements into one statement which will return `"Yes"` if `val` is less than or equal to `50` and greater than or equal to `25` . Otherwise, will return `"No"` .

Challenge Seed

```
function testLogicalAnd(val) {
  // Only change code below this line

  if (val) {
    if (val) {
      return "Yes";
    }
  }

  // Only change code above this line
  return "No";
}

// Change this value to test
testLogicalAnd(10);
```

Solution

```
function testLogicalAnd(val) {
  if (val >= 25 && val <= 50) {
    return "Yes";
  }
  return "No";
}
```

67. Comparisons with the Logical Or Operator

Description

The logical or operator (`||`) returns `true` if either of the operands is `true` . Otherwise, it returns `false` . The logical or operator is composed of two pipe symbols (`|`). This can typically be found between your Backspace and Enter keys. The pattern below should look familiar from prior waypoints:

```
if (num > 10) {
  return "No";
}
if (num < 5) {
  return "No";
```

```
}
return "Yes";
```

will return "Yes" only if `num` is between `5` and `10` (5 and 10 included). The same logic can be written as:

```
if (num > 10 || num < 5) {
  return "No";
}
return "Yes";
```

Instructions

Combine the two `if` statements into one statement which returns `"Outside"` if `val` is not between `10` and `20`, inclusive. Otherwise, return `"Inside"`.

Challenge Seed

```
function testLogicalOr(val) {
  // Only change code below this line

  if (val) {
    return "Outside";
  }

  if (val) {
    return "Outside";
  }

  // Only change code above this line
  return "Inside";
}

// Change this value to test
testLogicalOr(15);
```

Solution

```
function testLogicalOr(val) {
  if (val < 10 || val > 20) {
    return "Outside";
  }
  return "Inside";
}
```

68. Introducing Else Statements

Description

When a condition for an `if` statement is true, the block of code following it is executed. What about when that condition is false? Normally nothing would happen. With an `else` statement, an alternate block of code can be executed.

```
if (num > 10) {
  return "Bigger than 10";
} else {
  return "10 or Less";
}
```

Instructions

Combine the `if` statements into a single `if/else` statement.

Challenge Seed

```
function testElse(val) {
  var result = "";
  // Only change code below this line

  if (val > 5) {
    result = "Bigger than 5";
  }

  if (val <= 5) {
    result = "5 or Smaller";
  }

  // Only change code above this line
  return result;
}

// Change this value to test
testElse(4);
```

Solution

```
function testElse(val) {
  var result = "";
  if(val > 5) {
    result = "Bigger than 5";
  } else {
    result = "5 or Smaller";
  }
  return result;
}
```

69. Introducing Else If Statements

Description

If you have multiple conditions that need to be addressed, you can chain `if` statements together with `else if` statements.

```
if (num > 15) {
  return "Bigger than 15";
} else if (num < 5) {
  return "Smaller than 5";
} else {
  return "Between 5 and 15";
}
```

Instructions

Convert the logic to use `else if` statements.

Challenge Seed

```
function testElseIf(val) {
  if (val > 10) {
    return "Greater than 10";
  }

  if (val < 5) {
    return "Smaller than 5";
  }
```

```
  return "Between 5 and 10";
}

// Change this value to test
testElseIf(7);
```

Solution

```
function testElseIf(val) {
  if(val > 10) {
    return "Greater than 10";
  } else if(val < 5) {
    return "Smaller than 5";
  } else {
    return "Between 5 and 10";
  }
}
```

70. Logical Order in If Else Statements

Description

Order is important in `if, else if` statements. The function is executed from top to bottom so you will want to be careful of what statement comes first. Take these two functions as an example. Here's the first:

```
function foo(x) {
  if (x < 1) {
    return "Less than one";
  } else if (x < 2) {
    return "Less than two";
  } else {
    return "Greater than or equal to two";
  }
}
```

And the second just switches the order of the statements:

```
function bar(x) {
  if (x < 2) {
    return "Less than two";
  } else if (x < 1) {
    return "Less than one";
  } else {
    return "Greater than or equal to two";
  }
}
```

While these two functions look nearly identical if we pass a number to both we get different outputs.

```
foo(0) // "Less than one"
bar(0) // "Less than two"
```

Instructions

Change the order of logic in the function so that it will return the correct statements in all cases.

Challenge Seed

```
function orderMyLogic(val) {
  if (val < 10) {
    return "Less than 10";
  } else if (val < 5) {
    return "Less than 5";
```

```
  } else {
    return "Greater than or equal to 10";
  }
}

// Change this value to test
orderMyLogic(7);
```

Solution

```
function orderMyLogic(val) {
  if(val < 5) {
    return "Less than 5";
  } else if (val < 10) {
    return "Less than 10";
  } else {
    return "Greater than or equal to 10";
  }
}
```

71. Chaining If Else Statements

Description

`if/else` statements can be chained together for complex logic. Here is pseudocode of multiple chained `if / else if` statements:

> if (*condition1*) {
> *statement1*
> } else if (*condition2*) {
> *statement2*
> } else if (*condition3*) {
> *statement3*
> ...
> } else {
> *statementN*
> }

Instructions

Write chained `if` / `else if` statements to fulfill the following conditions: `num < 5` - return "Tiny"
`num < 10` - return "Small"
`num < 15` - return "Medium"
`num < 20` - return "Large"
`num >= 20` - return "Huge"

Challenge Seed

```
function testSize(num) {
  // Only change code below this line


  return "Change Me";
  // Only change code above this line
}

// Change this value to test
testSize(7);
```

Solution

```
function testSize(num) {
  if (num < 5) {
    return "Tiny";
  } else if (num < 10) {
    return "Small";
  } else if (num < 15) {
    return "Medium";
  } else if (num < 20) {
    return "Large";
  } else {
    return "Huge";
  }
}
```

72. Golf Code

Description

In the game of golf each hole has a `par` meaning the average number of `strokes` a golfer is expected to make in order to sink the ball in a hole to complete the play. Depending on how far above or below `par` your `strokes` are, there is a different nickname. Your function will be passed `par` and `strokes` arguments. Return the correct string according to this table which lists the strokes in order of priority; top (highest) to bottom (lowest):

Strokes	Return
1	"Hole-in-one!"
<= par - 2	"Eagle"
par - 1	"Birdie"
par	"Par"
par + 1	"Bogey"
par + 2	"Double Bogey"
>= par + 3	"Go Home!"

`par` and `strokes` will always be numeric and positive. We have added an array of all the names for your convenience.

Instructions

Challenge Seed

```
var names = ["Hole-in-one!", "Eagle", "Birdie", "Par", "Bogey", "Double Bogey", "Go Home!"];
function golfScore(par, strokes) {
  // Only change code below this line


  return "Change Me";
  // Only change code above this line
}

// Change these values to test
golfScore(5, 4);
```

Solution

```
function golfScore(par, strokes) {
  if (strokes === 1) {
    return "Hole-in-one!";
  }
```

```
  if (strokes <= par - 2) {
    return "Eagle";
  }

  if (strokes === par - 1) {
    return "Birdie";
  }

  if (strokes === par) {
    return "Par";
  }

  if (strokes === par + 1) {
    return "Bogey";
  }

  if(strokes === par + 2) {
    return "Double Bogey";
  }

  return "Go Home!";
}
```

73. Selecting from Many Options with Switch Statements

Description

If you have many options to choose from, use a `switch` statement. A `switch` statement tests a value and can have many `case` statements which define various possible values. Statements are executed from the first matched `case` value until a `break` is encountered. Here is a pseudocode example:

```
switch(num) {
  case value1:
    statement1;
    break;
  case value2:
    statement2;
    break;
...
  case valueN:
    statementN;
    break;
}
```

`case` values are tested with strict equality (`===`). The `break` tells JavaScript to stop executing statements. If the `break` is omitted, the next statement will be executed.

Instructions

Write a switch statement which tests `val` and sets `answer` for the following conditions:
`1` - "alpha"
`2` - "beta"
`3` - "gamma"
`4` - "delta"

Challenge Seed

```
function caseInSwitch(val) {
  var answer = "";
  // Only change code below this line
```

```
  // Only change code above this line
  return answer;
}

// Change this value to test
caseInSwitch(1);
```

Solution

```
function caseInSwitch(val) {
  var answer = "";

  switch(val) {
    case 1:
      answer = "alpha";
      break;
    case 2:
      answer = "beta";
      break;
    case 3:
      answer = "gamma";
      break;
    case 4:
      answer = "delta";
  }
  return answer;
}
```

74. Adding a Default Option in Switch Statements

Description

In a `switch` statement you may not be able to specify all possible values as `case` statements. Instead, you can add the `default` statement which will be executed if no matching `case` statements are found. Think of it like the final `else` statement in an `if/else` chain. A `default` statement should be the last case.

```
switch (num) {
  case value1:
    statement1;
    break;
  case value2:
    statement2;
    break;
...
  default:
    defaultStatement;
    break;
}
```

Instructions

Write a switch statement to set `answer` for the following conditions:
`"a"` - "apple"
`"b"` - "bird"
`"c"` - "cat"
`default` - "stuff"

Challenge Seed

```
function switchOfStuff(val) {
  var answer = "";
  // Only change code below this line
```

```
  // Only change code above this line
  return answer;
}

// Change this value to test
switchOfStuff(1);
```

Solution

```
function switchOfStuff(val) {
  var answer = "";

  switch(val) {
    case "a":
      answer = "apple";
      break;
    case "b":
      answer = "bird";
      break;
    case "c":
      answer = "cat";
      break;
    default:
      answer = "stuff";
  }
  return answer;
}
```

75. Multiple Identical Options in Switch Statements

Description

If the `break` statement is omitted from a `switch` statement's `case`, the following `case` statement(s) are executed until a `break` is encountered. If you have multiple inputs with the same output, you can represent them in a `switch` statement like this:

```
switch(val) {
  case 1:
  case 2:
  case 3:
    result = "1, 2, or 3";
    break;
  case 4:
    result = "4 alone";
}
```

Cases for 1, 2, and 3 will all produce the same result.

Instructions

Write a switch statement to set `answer` for the following ranges:
`1-3` - "Low"
`4-6` - "Mid"
`7-9` - "High" **Note**
You will need to have a `case` statement for each number in the range.

Challenge Seed

```
function sequentialSizes(val) {
  var answer = "";
  // Only change code below this line



  // Only change code above this line
  return answer;
}

// Change this value to test
sequentialSizes(1);
```

Solution

```
function sequentialSizes(val) {
  var answer = "";

  switch(val) {
    case 1:
    case 2:
    case 3:
      answer = "Low";
      break;
    case 4:
    case 5:
    case 6:
      answer = "Mid";
      break;
    case 7:
    case 8:
    case 9:
      answer = "High";
  }

  return answer;
}
```

76. Replacing If Else Chains with Switch

Description

If you have many options to choose from, a `switch` statement can be easier to write than many chained `if` / `else if` statements. The following:

```
if (val === 1) {
  answer = "a";
} else if (val === 2) {
  answer = "b";
} else {
  answer = "c";
}
```

can be replaced with:

```
switch(val) {
  case 1:
    answer = "a";
    break;
  case 2:
    answer = "b";
    break;
  default:
    answer = "c";
}
```

Instructions

Change the chained `if` / `else if` statements into a `switch` statement.

Challenge Seed

```
function chainToSwitch(val) {
  var answer = "";
  // Only change code below this line

  if (val === "bob") {
    answer = "Marley";
  } else if (val === 42) {
    answer = "The Answer";
  } else if (val === 1) {
    answer = "There is no #1";
  } else if (val === 99) {
    answer = "Missed me by this much!";
  } else if (val === 7) {
    answer = "Ate Nine";
  }

  // Only change code above this line
  return answer;
}

// Change this value to test
chainToSwitch(7);
```

Solution

```
function chainToSwitch(val) {
  var answer = "";

  switch(val) {
    case "bob":
      answer = "Marley";
      break;
    case 42:
      answer = "The Answer";
      break;
    case 1:
      answer = "There is no #1";
      break;
    case 99:
      answer = "Missed me by this much!";
      break;
    case 7:
      answer = "Ate Nine";
  }
  return answer;
}
```

77. Returning Boolean Values from Functions

Description

You may recall from Comparison with the Equality Operator that all comparison operators return a boolean `true` or `false` value. Sometimes people use an if/else statement to do a comparison, like this:

```
function isEqual(a,b) {
  if (a === b) {
    return true;
  } else {
    return false;
```

```
  }
}
```

But there's a better way to do this. Since `===` returns `true` or `false`, we can return the result of the comparison:

```
function isEqual(a,b) {
  return a === b;
}
```

Instructions

Fix the function `isLess` to remove the `if/else` statements.

Challenge Seed

```
function isLess(a, b) {
  // Fix this code
  if (a < b) {
    return true;
  } else {
    return false;
  }
}

// Change these values to test
isLess(10, 15);
```

Solution

```
function isLess(a, b) {
  return a < b;
}
```

78. Return Early Pattern for Functions

Description

When a `return` statement is reached, the execution of the current function stops and control returns to the calling location. **Example**

```
function myFun() {
  console.log("Hello");
  return "World";
  console.log("byebye")
}
myFun();
```

The above outputs "Hello" to the console, returns "World", but `"byebye"` is never output, because the function exits at the `return` statement.

Instructions

Modify the function `abTest` so that if `a` or `b` are less than `0` the function will immediately exit with a value of `undefined`. **Hint**
Remember that `undefined` is a keyword, not a string.

Challenge Seed

```
// Setup
function abTest(a, b) {
```

```
  // Only change code below this line



  // Only change code above this line

  return Math.round(Math.pow(Math.sqrt(a) + Math.sqrt(b), 2));
}

// Change values below to test your code
abTest(2,2);
```

Solution

```
function abTest(a, b) {
  if(a < 0 || b < 0) {
    return undefined;
  }
  return Math.round(Math.pow(Math.sqrt(a) + Math.sqrt(b), 2));
}
```

79. Counting Cards

Description

In the casino game Blackjack, a player can gain an advantage over the house by keeping track of the relative number of high and low cards remaining in the deck. This is called Card Counting. Having more high cards remaining in the deck favors the player. Each card is assigned a value according to the table below. When the count is positive, the player should bet high. When the count is zero or negative, the player should bet low.

Count Change	Cards
+1	2, 3, 4, 5, 6
0	7, 8, 9
-1	10, 'J', 'Q', 'K', 'A'

You will write a card counting function. It will receive a `card` parameter, which can be a number or a string, and increment or decrement the global `count` variable according to the card's value (see table). The function will then return a string with the current count and the string `Bet` if the count is positive, or `Hold` if the count is zero or negative. The current count and the player's decision (`Bet` or `Hold`) should be separated by a single space. **Example Output**
`-3 Hold`
`5 Bet` **Hint**
Do NOT reset `count` to 0 when value is 7, 8, or 9.
Do NOT return an array.
Do NOT include quotes (single or double) in the output.

Instructions

Challenge Seed

```
var count = 0;

function cc(card) {
  // Only change code below this line


  return "Change Me";
  // Only change code above this line
}
```

```
// Add/remove calls to test your function.
// Note: Only the last will display
cc(2); cc(3); cc(7); cc('K'); cc('A');
```

Solution

```
var count = 0;
function cc(card) {
  switch(card) {
    case 2:
    case 3:
    case 4:
    case 5:
    case 6:
      count++;
      break;
    case 10:
    case 'J':
    case 'Q':
    case 'K':
    case 'A':
      count--;
  }
  if(count > 0) {
    return count + " Bet";
  } else {
    return count + " Hold";
  }
}
```

80. Build JavaScript Objects

Description

You may have heard the term `object` before. 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. Here's a sample cat object:

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

In this example, all the properties are stored as strings, such as - `"name"`, `"legs"`, and `"tails"`. However, you can also use numbers as properties. You can even omit the quotes for single-word string properties, as follows:

```
var anotherObject = {
  make: "Ford",
  5: "five",
  "model": "focus"
};
```

However, if your object has any non-string properties, JavaScript will automatically typecast them as strings.

Instructions

Make an object that represents a dog called `myDog` which contains the properties `"name"` (a string), `"legs"`, `"tails"` and `"friends"`. You can set these object properties to whatever values you want, as long `"name"` is a string, `"legs"` and `"tails"` are numbers, and `"friends"` is an array.

Challenge Seed

```
// Example
var ourDog = {
  "name": "Camper",
  "legs": 4,
  "tails": 1,
  "friends": ["everything!"]
};

// Only change code below this line.

var myDog = {




};
```

After Test

```
(function(z){return z;})(myDog);
```

Solution

```
var myDog = {
  "name": "Camper",
  "legs": 4,
  "tails": 1,
  "friends": ["everything!"]
};
```

81. Accessing Object Properties with Dot Notation

Description

There are two ways to access the properties of an object: dot notation (`.`) and bracket notation (`[]`), similar to an array. Dot notation is what you use when you know the name of the property you're trying to access ahead of time. Here is a sample of using dot notation (`.`) to read an object's property:

```
var myObj = {
  prop1: "val1",
  prop2: "val2"
};
var prop1val = myObj.prop1; // val1
var prop2val = myObj.prop2; // val2
```

Instructions

Read in the property values of `testObj` using dot notation. Set the variable `hatValue` equal to the object's property `hat` and set the variable `shirtValue` equal to the object's property `shirt`.

Challenge Seed

```
// Setup
var testObj = {
  "hat": "ballcap",
  "shirt": "jersey",
  "shoes": "cleats"
};

// Only change code below this line
```

```
var hatValue = testObj;      // Change this line
var shirtValue = testObj;    // Change this line
```

After Test

```
(function(a,b) { return "hatValue = '" + a + "', shirtValue = '" + b + "'"; })(hatValue,shirtValue);
```

Solution

```
var testObj = {
  "hat": "ballcap",
  "shirt": "jersey",
  "shoes": "cleats"
};

var hatValue = testObj.hat;
var shirtValue = testObj.shirt;
```

82. Accessing Object Properties with Bracket Notation

Description

The second way to access the properties of an object is bracket notation (`[]`). If the property of the object you are trying to access has a space in its name, you will need to use bracket notation. However, you can still use bracket notation on object properties without spaces. Here is a sample of using bracket notation to read an object's property:

```
var myObj = {
  "Space Name": "Kirk",
  "More Space": "Spock",
  "NoSpace": "USS Enterprise"
};
myObj["Space Name"]; // Kirk
myObj['More Space']; // Spock
myObj["NoSpace"]; // USS Enterprise
```

Note that property names with spaces in them must be in quotes (single or double).

Instructions

Read the values of the properties `"an entree"` and `"the drink"` of `testObj` using bracket notation and assign them to `entreeValue` and `drinkValue` respectively.

Challenge Seed

```
// Setup
var testObj = {
  "an entree": "hamburger",
  "my side": "veggies",
  "the drink": "water"
};

// Only change code below this line

var entreeValue = testObj;   // Change this line
var drinkValue = testObj;    // Change this line
```

After Test

```
(function(a,b) { return "entreeValue = '" + a + "', drinkValue = '" + b + "'"; })
(entreeValue,drinkValue);
```

Solution

```
var testObj = {
  "an entree": "hamburger",
  "my side": "veggies",
  "the drink": "water"
};
var entreeValue = testObj["an entree"];
var drinkValue = testObj['the drink'];
```

83. Accessing Object Properties with Variables

Description

Another use of bracket notation on objects is to access a property which is stored as the value of a variable. This can be very useful for iterating through an object's properties or when accessing a lookup table. Here is an example of using a variable to access a property:

```
var dogs = {
  Fido: "Mutt", Hunter: "Doberman", Snoopie: "Beagle"
};
var myDog = "Hunter";
var myBreed = dogs[myDog];
console.log(myBreed); // "Doberman"
```

Another way you can use this concept is when the property's name is collected dynamically during the program execution, as follows:

```
var someObj = {
  propName: "John"
};
function propPrefix(str) {
  var s = "prop";
  return s + str;
}
var someProp = propPrefix("Name"); // someProp now holds the value 'propName'
console.log(someObj[someProp]); // "John"
```

Note that we do *not* use quotes around the variable name when using it to access the property because we are using the *value* of the variable, not the *name*.

Instructions

Use the `playerNumber` variable to look up player `16` in `testObj` using bracket notation. Then assign that name to the `player` variable.

Challenge Seed

```
// Setup
var testObj = {
  12: "Namath",
  16: "Montana",
  19: "Unitas"
};

// Only change code below this line;
```

```
var playerNumber;       // Change this Line
var player = testObj;   // Change this Line
```

After Test

```
if(typeof player !== "undefined"){(function(v){return v;})(player);}
```

Solution

```
var testObj = {
  12: "Namath",
  16: "Montana",
  19: "Unitas"
};
var playerNumber = 16;
var player = testObj[playerNumber];
```

84. Updating Object Properties

Description

After you've created a JavaScript object, you can update its properties at any time just like you would update any other variable. You can use either dot or bracket notation to update. For example, let's look at `ourDog`:

> var ourDog = {
> "name": "Camper",
> "legs": 4,
> "tails": 1,
> "friends": ["everything!"]
> };

Since he's a particularly happy dog, let's change his name to "Happy Camper". Here's how we update his object's name property: `ourDog.name = "Happy Camper";` or `ourDog["name"] = "Happy Camper";` Now when we evaluate `ourDog.name`, instead of getting "Camper", we'll get his new name, "Happy Camper".

Instructions

Update the `myDog` object's name property. Let's change her name from "Coder" to "Happy Coder". You can use either dot or bracket notation.

Challenge Seed

```
// Example
var ourDog = {
  "name": "Camper",
  "legs": 4,
  "tails": 1,
  "friends": ["everything!"]
};

ourDog.name = "Happy Camper";

// Setup
var myDog = {
  "name": "Coder",
  "legs": 4,
  "tails": 1,
  "friends": ["freeCodeCamp Campers"]
};

// Only change code below this line.
```

After Test

```
(function(z){return z;})(myDog);
```

Solution

```
var myDog = {
  "name": "Coder",
  "legs": 4,
  "tails": 1,
  "friends": ["freeCodeCamp Campers"]
};
myDog.name = "Happy Coder";
```

85. Add New Properties to a JavaScript Object

Description

You can add new properties to existing JavaScript objects the same way you would modify them. Here's how we would add a `"bark"` property to `ourDog` : `ourDog.bark = "bow-wow";` or `ourDog["bark"] = "bow-wow";` Now when we evaluate `ourDog.bark` , we'll get his bark, "bow-wow".

Instructions

Add a `"bark"` property to `myDog` and set it to a dog sound, such as "woof". You may use either dot or bracket notation.

Challenge Seed

```
// Example
var ourDog = {
  "name": "Camper",
  "legs": 4,
  "tails": 1,
  "friends": ["everything!"]
};

ourDog.bark = "bow-wow";

// Setup
var myDog = {
  "name": "Happy Coder",
  "legs": 4,
  "tails": 1,
  "friends": ["freeCodeCamp Campers"]
};

// Only change code below this line.
```

After Test

```
(function(z){return z;})(myDog);
```

Solution

```
var myDog = {
  "name": "Happy Coder",
```

```
  "legs": 4,
  "tails": 1,
  "friends": ["freeCodeCamp Campers"]
};
myDog.bark = "Woof Woof";
```

86. Delete Properties from a JavaScript Object

Description

We can also delete properties from objects like this: `delete ourDog.bark;`

Instructions

Delete the `"tails"` property from `myDog`. You may use either dot or bracket notation.

Challenge Seed

```
// Example
var ourDog = {
  "name": "Camper",
  "legs": 4,
  "tails": 1,
  "friends": ["everything!"],
  "bark": "bow-wow"
};

delete ourDog.bark;

// Setup
var myDog = {
  "name": "Happy Coder",
  "legs": 4,
  "tails": 1,
  "friends": ["freeCodeCamp Campers"],
  "bark": "woof"
};

// Only change code below this line.
```

After Test

```
(function(z){return z;})(myDog);
```

Solution

```
var ourDog = {
  "name": "Camper",
  "legs": 4,
  "tails": 1,
  "friends": ["everything!"],
  "bark": "bow-wow"
};
var myDog = {
  "name": "Happy Coder",
  "legs": 4,
  "tails": 1,
  "friends": ["freeCodeCamp Campers"],
  "bark": "woof"
};
delete myDog.tails;
```

87. Using Objects for Lookups

Description

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. Here is an example of a simple reverse alphabet lookup:

```
var alpha = {
  1:"Z",
  2:"Y",
  3:"X",
  4:"W",
  ...
  24:"C",
  25:"B",
  26:"A"
};
alpha[2]; // "Y"
alpha[24]; // "C"

var value = 2;
alpha[value]; // "Y"
```

Instructions

Convert the switch statement into an object called `lookup`. Use it to look up `val` and assign the associated string to the `result` variable.

Challenge Seed

```
// Setup
function phoneticLookup(val) {
  var result = "";

  // Only change code below this line
  switch(val) {
    case "alpha":
      result = "Adams";
      break;
    case "bravo":
      result = "Boston";
      break;
    case "charlie":
      result = "Chicago";
      break;
    case "delta":
      result = "Denver";
      break;
    case "echo":
      result = "Easy";
      break;
    case "foxtrot":
      result = "Frank";
  }

  // Only change code above this line
  return result;
}

// Change this value to test
phoneticLookup("charlie");
```

Solution

```
function phoneticLookup(val) {
  var result = "";

  var lookup = {
    alpha: "Adams",
    bravo: "Boston",
    charlie: "Chicago",
    delta: "Denver",
    echo: "Easy",
    foxtrot: "Frank"
  };

  result = lookup[val];

  return result;
}
```

88. Testing Objects for Properties

Description

Sometimes it is useful to check if the property of a given object exists or not. We can use the `.hasOwnProperty(propname)` method of objects to determine if that object has the given property name. `.hasOwnProperty()` returns `true` or `false` if the property is found or not. **Example**

```
var myObj = {
  top: "hat",
  bottom: "pants"
};
myObj.hasOwnProperty("top"); // true
myObj.hasOwnProperty("middle"); // false
```

Instructions

Modify the function `checkObj` to test `myObj` for `checkProp`. If the property is found, return that property's value. If not, return `"Not Found"`.

Challenge Seed

```
// Setup
var myObj = {
  gift: "pony",
  pet: "kitten",
  bed: "sleigh"
};

function checkObj(checkProp) {
  // Your Code Here

  return "Change Me!";
}

// Test your code by modifying these values
checkObj("gift");
```

Solution

```
var myObj = {
  gift: "pony",
  pet: "kitten",
  bed: "sleigh"
};
```

```
function checkObj(checkProp) {
  if(myObj.hasOwnProperty(checkProp)) {
    return myObj[checkProp];
  } else {
    return "Not Found";
  }
}
```

89. Manipulating Complex Objects

Description

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 or `JSON` is a related data interchange format used to store data.

```
{
 "artist": "Daft Punk",
 "title": "Homework",
 "release_year": 1997,
 "formats": [
  "CD",
  "Cassette",
  "LP"
 ],
 "gold": true
}
```

Note
You will need to place a comma after every object in the array, unless it is the last object in the array.

Instructions

Add a new album to the `myMusic` array. Add `artist` and `title` strings, `release_year` number, and a `formats` array of strings.

Challenge Seed

```
var myMusic = [
  {
    "artist": "Billy Joel",
    "title": "Piano Man",
```

```
    "release_year": 1973,
    "formats": [
      "CD",
      "8T",
      "LP"
    ],
    "gold": true
  }
  // Add record here
];
```

After Test

```
(function(x){ if (Array.isArray(x)) { return JSON.stringify(x); } return "myMusic is not an array"})
(myMusic);
```

Solution

```
var myMusic = [
  {
    "artist": "Billy Joel",
    "title": "Piano Man",
    "release_year": 1973,
    "formats": [
      "CS",
      "8T",
      "LP" ],
    "gold": true
  },
  {
    "artist": "ABBA",
    "title": "Ring Ring",
    "release_year": 1973,
    "formats": [
      "CS",
      "8T",
      "LP",
    "CD",
  ]
  }
];
```

90. Accessing Nested Objects

Description

The sub-properties of objects can be accessed by chaining together the dot or bracket notation. Here is a nested object:

```
var ourStorage = {
 "desk": {
  "drawer": "stapler"
 },
 "cabinet": {
  "top drawer": {
   "folder1": "a file",
   "folder2": "secrets"
  },
  "bottom drawer": "soda"
 }
};
ourStorage.cabinet["top drawer"].folder2; // "secrets"
ourStorage.desk.drawer; // "stapler"
```

Instructions

Access the `myStorage` object and assign the contents of the `glove box` property to the `gloveBoxContents` variable. Use bracket notation for properties with a space in their name.

Challenge Seed

```
// Setup
var myStorage = {
  "car": {
    "inside": {
      "glove box": "maps",
      "passenger seat": "crumbs"
     },
    "outside": {
      "trunk": "jack"
    }
  }
};

var gloveBoxContents = undefined; // Change this line
```

After Test

```
(function(x) {
  if(typeof x != 'undefined') {
    return "gloveBoxContents = " + x;
  }
  return "gloveBoxContents is undefined";
})(gloveBoxContents);
```

Solution

```
var myStorage = {
  "car":{
    "inside":{
      "glove box":"maps",
      "passenger seat":"crumbs"
    },
    "outside":{
      "trunk":"jack"
    }
  }
};
var gloveBoxContents = myStorage.car.inside["glove box"];
```

91. Accessing Nested Arrays

Description

As we have seen in earlier examples, objects can contain both nested objects and nested arrays. Similar to accessing nested objects, Array bracket notation can be chained to access nested arrays. Here is an example of how to access a nested array:

```
var ourPets = [
 {
  animalType: "cat",
  names: [
   "Meowzer",
   "Fluffy",
   "Kit-Cat"
  ]
```

```
  },
  {
   animalType: "dog",
   names: [
    "Spot",
    "Bowser",
    "Frankie"
   ]
  }
];
ourPets[0].names[1]; // "Fluffy"
ourPets[1].names[0]; // "Spot"
```

Instructions

Retrieve the second tree from the variable `myPlants` using object dot and array bracket notation.

Challenge Seed

```
// Setup
var myPlants = [
  {
    type: "flowers",
    list: [
      "rose",
      "tulip",
      "dandelion"
    ]
  },
  {
    type: "trees",
    list: [
      "fir",
      "pine",
      "birch"
    ]
  }
];

// Only change code below this line

var secondTree = ""; // Change this line
```

After Test

```
(function(x) {
  if(typeof x != 'undefined') {
    return "secondTree = " + x;
  }
  return "secondTree is undefined";
})(secondTree);
```

Solution

```
var myPlants = [
  {
    type: "flowers",
    list: [
      "rose",
      "tulip",
      "dandelion"
    ]
  },
  {
    type: "trees",
    list: [
```

```
      "fir",
      "pine",
      "birch"
    ]
  }
];

// Only change code below this line

var secondTree = myPlants[1].list[1];
```

92. Record Collection

Description

You are given a JSON object representing a part of your musical album collection. Each album has several properties and a unique id number as its key. Not all albums have complete information. Write a function which takes an album's `id` (like `2548`), a property `prop` (like `"artist"` or `"tracks"`), and a `value` (like `"Addicted to Love"`) to modify the data in this collection. If `prop` isn't `"tracks"` and `value` isn't empty (`""`), update or set the `value` for that record album's property. Your function must always return the entire collection object. There are several rules for handling incomplete data: If `prop` is `"tracks"` but the album doesn't have a `"tracks"` property, create an empty array before adding the new value to the album's corresponding property. If `prop` is `"tracks"` and `value` isn't empty (`""`), push the `value` onto the end of the album's existing `tracks` array. If `value` is empty (`""`), delete the given `prop` property from the album. **Hints**
Use `bracket notation` when accessing object properties with variables. Push is an array method you can read about on Mozilla Developer Network. You may refer back to Manipulating Complex Objects Introducing JavaScript Object Notation (JSON) for a refresher.

Instructions

Challenge Seed

```
// Setup
var collection = {
    "2548": {
      "album": "Slippery When Wet",
      "artist": "Bon Jovi",
      "tracks": [
        "Let It Rock",
        "You Give Love a Bad Name"
      ]
    },
    "2468": {
      "album": "1999",
      "artist": "Prince",
      "tracks": [
        "1999",
        "Little Red Corvette"
      ]
    },
    "1245": {
      "artist": "Robert Palmer",
      "tracks": [ ]
    },
    "5439": {
      "album": "ABBA Gold"
    }
};
// Keep a copy of the collection for tests
var collectionCopy = JSON.parse(JSON.stringify(collection));

// Only change code below this line
function updateRecords(id, prop, value) {
```

```
  return collection;
}

// Alter values below to test your code
updateRecords(5439, "artist", "ABBA");
```

After Test

```
;(function(x) { return "collection = \n" + JSON.stringify(x, '\n', 2); })(collection);
```

Solution

```
var collection = {
    2548: {
      album: "Slippery When Wet",
      artist: "Bon Jovi",
      tracks: [
        "Let It Rock",
        "You Give Love a Bad Name"
      ]
    },
    2468: {
      album: "1999",
      artist: "Prince",
      tracks: [
        "1999",
        "Little Red Corvette"
      ]
    },
    1245: {
      artist: "Robert Palmer",
      tracks: [ ]
    },
    5439: {
      album: "ABBA Gold"
    }
};
// Keep a copy of the collection for tests
var collectionCopy = JSON.parse(JSON.stringify(collection));

// Only change code below this line
function updateRecords(id, prop, value) {
  if(value === "") delete collection[id][prop];
  else if(prop === "tracks") {
    collection[id][prop] = collection[id][prop] || [];
    collection[id][prop].push(value);
  } else {
    collection[id][prop] = value;
  }

  return collection;
}
```

93. Iterate with JavaScript While Loops

Description

You can run the same code multiple times by using a loop. 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++;
}
```

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

Instructions

Push the numbers 0 through 4 to `myArray` using a `while` loop.

Challenge Seed

```
// Setup
var myArray = [];

// Only change code below this line.
```

After Test

```
if(typeof myArray !== "undefined"){(function(){return myArray;})();}
```

Solution

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

94. Iterate with JavaScript For Loops

Description

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])` The `initialization` statement is executed one time only before the loop starts. It is typically used to define and setup your loop variable. The `condition` statement is evaluated at the beginning of every loop iteration and will continue as long as it evaluates to `true` . When `condition` is `false` at the start of the iteration, the loop will stop executing. This means if `condition` starts as `false` , your loop will never execute. The `final-expression` is executed at the end of each loop iteration, prior to the next `condition` check and is usually used to increment or decrement your loop counter. In the following example we initialize with `i = 0` and iterate while our condition `i < 5` is true. We'll increment `i` by `1` in each loop iteration with `i++` as our `final-expression` .

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

`ourArray` will now contain `[0,1,2,3,4]` .

Instructions

Use a `for` loop to work to push the values 1 through 5 onto `myArray` .

Challenge Seed

```
// Example
var ourArray = [];
```

```
for (var i = 0; i < 5; i++) {
  ourArray.push(i);
}

// Setup
var myArray = [];

// Only change code below this line.
```

After Test

```
if (typeof myArray !== "undefined"){(function(){return myArray;})();}
```

Solution

```
var ourArray = [];
for (var i = 0; i < 5; i++) {
  ourArray.push(i);
}
var myArray = [];
for (var i = 1; i < 6; i++) {
  myArray.push(i);
}
```

95. Iterate Odd Numbers With a For Loop

Description

For loops don't have to iterate one at a time. By changing our `final-expression`, we can count by even numbers. We'll start at `i = 0` and loop while `i < 10`. We'll increment `i` by 2 each loop with `i += 2`.

> var ourArray = [];
> for (var i = 0; i < 10; i += 2) {
> ourArray.push(i);
> }

`ourArray` will now contain `[0,2,4,6,8]`. Let's change our `initialization` so we can count by odd numbers.

Instructions

Push the odd numbers from 1 through 9 to `myArray` using a `for` loop.

Challenge Seed

```
// Example
var ourArray = [];

for (var i = 0; i < 10; i += 2) {
  ourArray.push(i);
}

// Setup
var myArray = [];

// Only change code below this line.
```

After Test

```
if(typeof myArray !== "undefined"){(function(){return myArray;})();}
```

Solution

```
var ourArray = [];
for (var i = 0; i < 10; i += 2) {
  ourArray.push(i);
}
var myArray = [];
for (var i = 1; i < 10; i += 2) {
  myArray.push(i);
}
```

96. Count Backwards With a For Loop

Description

A for loop can also count backwards, so long as we can define the right conditions. In order to count backwards by twos, we'll need to change our `initialization`, `condition`, and `final-expression`. We'll start at `i = 10` and loop while `i > 0`. We'll decrement `i` by 2 each loop with `i -= 2`.

```
var ourArray = [];
for (var i=10; i > 0; i-=2) {
  ourArray.push(i);
}
```

`ourArray` will now contain `[10,8,6,4,2]`. Let's change our `initialization` and `final-expression` so we can count backward by twos by odd numbers.

Instructions

Push the odd numbers from 9 through 1 to `myArray` using a `for` loop.

Challenge Seed

```
// Example
var ourArray = [];

for (var i = 10; i > 0; i -= 2) {
  ourArray.push(i);
}

// Setup
var myArray = [];

// Only change code below this line.
```

After Test

```
if(typeof myArray !== "undefined"){(function(){return myArray;})();}
```

Solution

```
var ourArray = [];
for (var i = 10; i > 0; i -= 2) {
  ourArray.push(i);
}
var myArray = [];
```

```
for (var i = 9; i > 0; i -= 2) {
  myArray.push(i);
}
```

97. Iterate Through an Array with a For Loop

Description

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.

Instructions

Declare and initialize a variable `total` to `0`. Use a `for` loop to add the value of each element of the `myArr` array to `total`.

Challenge Seed

```
// Example
var ourArr = [ 9, 10, 11, 12];
var ourTotal = 0;

for (var i = 0; i < ourArr.length; i++) {
  ourTotal += ourArr[i];
}

// Setup
var myArr = [ 2, 3, 4, 5, 6];

// Only change code below this line
```

After Test

```
(function(){if(typeof total !== 'undefined') { return "total = " + total; } else { return "total is
undefined";}})()
```

Solution

```
var ourArr = [ 9, 10, 11, 12];
var ourTotal = 0;

for (var i = 0; i < ourArr.length; i++) {
  ourTotal += ourArr[i];
}

var myArr = [ 2, 3, 4, 5, 6];
var total = 0;

for (var i = 0; i < myArr.length; i++) {
  total += myArr[i];
}
```

98. Nesting For Loops

Description

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.

Instructions

Modify function `multiplyAll` so that it multiplies the `product` variable by each number in the sub-arrays of `arr`

Challenge Seed

```
function multiplyAll(arr) {
  var product = 1;
  // Only change code below this line

  // Only change code above this line
  return product;
}

// Modify values below to test your code
multiplyAll([[1,2],[3,4],[5,6,7]]);
```

Solution

```
function multiplyAll(arr) {
  var product = 1;
  for (var i = 0; i < arr.length; i++) {
    for (var j = 0; j < arr[i].length; j++) {
      product *= arr[i][j];
    }
  }
  return product;
}

multiplyAll([[1,2],[3,4],[5,6,7]]);
```

99. Iterate with JavaScript Do...While Loops

Description

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. Here is a regular while loop that will run the code in the loop as long as `i < 5`.

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

Notice that we initialize the value of `i` to be 5. When we execute the next line, we notice that `i` is not less than 5. So we do not execute the code inside the loop. The result is that `ourArray` will end up with nothing added to it, so it will still look like this `[]` when all the code in the example above finishes running. Now, take a look at a `do...while` loop.

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

In this case, we initialize the value of `i` as 5, just like we did with the while loop. When we get to the next line, there is no check for the value of `i`, so we go to the code inside the curly braces and execute it. We will add one element to the array and increment `i` before we get to the condition check. Then, when we get to checking if `i < 5` see that `i` is now 6, which fails the conditional check. So we exit the loop and are done. At the end of the above example, the value of `ourArray` is `[5]`. 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.

Instructions

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

Challenge Seed

```
// Setup
var myArray = [];
var i = 10;

// Only change code below this line.

while (i < 5) {
  myArray.push(i);
  i++;
}
```

After Test

```
if(typeof myArray !== "undefined"){(function(){return myArray;})();}
```

Solution

```
var myArray = [];
var i = 10;
do {
  myArray.push(i);
  i++;
} while (i < 5)
```

100. Profile Lookup

Description

We have an array of objects representing different people in our contacts lists. A `lookUpProfile` function that takes `name` and a property (`prop`) as arguments has been pre-written for you. The function should check if `name` is an actual contact's `firstName` and the given property (`prop`) is a property of that contact. If both are true, then return the "value" of that property. If `name` does not correspond to any contacts then return `"No such contact"` If `prop` does not correspond to any valid properties of a contact found to match `name` then return `"No such property"`

Instructions

Challenge Seed

```
//Setup
var contacts = [
    {
        "firstName": "Akira",
        "lastName": "Laine",
        "number": "0543236543",
        "likes": ["Pizza", "Coding", "Brownie Points"]
    },
    {
        "firstName": "Harry",
        "lastName": "Potter",
        "number": "0994372684",
        "likes": ["Hogwarts", "Magic", "Hagrid"]
    },
    {
        "firstName": "Sherlock",
        "lastName": "Holmes",
        "number": "0487345643",
        "likes": ["Intriguing Cases", "Violin"]
    },
    {
        "firstName": "Kristian",
        "lastName": "Vos",
        "number": "unknown",
        "likes": ["JavaScript", "Gaming", "Foxes"]
    }
];


function lookUpProfile(name, prop){
// Only change code below this line

// Only change code above this line
}

// Change these values to test your function
lookUpProfile("Akira", "likes");
```

Solution

```
var contacts = [
    {
        "firstName": "Akira",
        "lastName": "Laine",
        "number": "0543236543",
        "likes": ["Pizza", "Coding", "Brownie Points"]
    },
    {
        "firstName": "Harry",
        "lastName": "Potter",
```

```
        "number": "0994372684",
        "likes": ["Hogwarts", "Magic", "Hagrid"]
    },
    {
        "firstName": "Sherlock",
        "lastName": "Holmes",
        "number": "0487345643",
        "likes": ["Intriguing Cases", "Violin"]
    },
    {
        "firstName": "Kristian",
        "lastName": "Vos",
        "number": "unknown",
        "likes": ["JavaScript", "Gaming", "Foxes"]
    },
];


//Write your function in between these comments
function lookUpProfile(name, prop){
    for(var i in contacts){
      if(contacts[i].firstName === name) {
        return contacts[i][prop] || "No such property";
      }
    }
   return "No such contact";
}
//Write your function in between these comments

lookUpProfile("Akira", "likes");
```

101. Generate Random Fractions with JavaScript

Description

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` **Note**
Like Storing Values with the Equal Operator, all function calls will be resolved before the `return` executes, so we can `return` the value of the `Math.random()` function.

Instructions

Change `randomFraction` to return a random number instead of returning `0`.

Challenge Seed

```
function randomFraction() {

  // Only change code below this line.

  return 0;

  // Only change code above this line.
}
```

After Test

```
(function(){return randomFraction();})();
```

Solution

```
function randomFraction() {
  return Math.random();
}
```

102. Generate Random Whole Numbers with JavaScript

Description

It's great that we can generate random decimal numbers, but it's even more useful if we use it to generate random whole numbers.

1. Use `Math.random()` to generate a random decimal.
2. Multiply that random decimal by `20`.
3. Use another function, `Math.floor()` to round the number down to its nearest whole number.

Remember that `Math.random()` can never quite return a `1` and, because we're rounding down, it's impossible to actually get `20`. This technique will give us a whole number between `0` and `19`. Putting everything together, this is what our code looks like: `Math.floor(Math.random() * 20);` We are calling `Math.random()`, multiplying the result by 20, then passing the value to `Math.floor()` function to round the value down to the nearest whole number.

Instructions

Use this technique to generate and return a random whole number between `0` and `9`.

Challenge Seed

```
var randomNumberBetween0and19 = Math.floor(Math.random() * 20);

function randomWholeNum() {

  // Only change code below this line.

  return Math.random();
}
```

After Test

```
(function(){return randomWholeNum();})();
```

Solution

```
var randomNumberBetween0and19 = Math.floor(Math.random() * 20);
function randomWholeNum() {
  return Math.floor(Math.random() * 10);
}
```

103. Generate Random Whole Numbers within a Range

Description

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`. Here's the formula we'll use. Take a moment to read it and try to understand what this code is doing: `Math.floor(Math.random() * (max - min + 1)) + min`

Instructions

Create a function called `randomRange` that takes a range `myMin` and `myMax` and returns a random number that's greater than or equal to `myMin`, and is less than or equal to `myMax`, inclusive.

Challenge Seed

```
// Example
function ourRandomRange(ourMin, ourMax) {

  return Math.floor(Math.random() * (ourMax - ourMin + 1)) + ourMin;
}

ourRandomRange(1, 9);

// Only change code below this line.

function randomRange(myMin, myMax) {

  return 0; // Change this line

}

// Change these values to test your function
var myRandom = randomRange(5, 15);
```

After Test

```
var calcMin = 100;
var calcMax = -100;
for(var i = 0; i < 100; i++) {
  var result = randomRange(5,15);
  calcMin = Math.min(calcMin, result);
  calcMax = Math.max(calcMax, result);
}
(function(){
  if(typeof myRandom === 'number') {
    return "myRandom = " + myRandom;
  } else {
    return "myRandom undefined";
  }
})()
```

Solution

```
function randomRange(myMin, myMax) {
  return Math.floor(Math.random() * (myMax - myMin + 1)) + myMin;
}
```

104. Use the parseInt Function

Description

The `parseInt()` function parses a string and returns an integer. Here's an example: `var a = parseInt("007");` The above function converts the string "007" to an integer 7. If the first character in the string can't be converted into a number, then it returns `NaN`.

Instructions

Use `parseInt()` in the `convertToInteger` function so it converts the input string `str` into an integer, and returns it.

Challenge Seed

```
function convertToInteger(str) {

}

convertToInteger("56");
```

Solution

```
function convertToInteger(str) {
  return parseInt(str);
}
```

105. Use the parseInt Function with a Radix

Description

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.

Instructions

Use `parseInt()` in the `convertToInteger` function so it converts a binary number to an integer and returns it.

Challenge Seed

```
function convertToInteger(str) {

}

convertToInteger("10011");
```

Solution

```
function convertToInteger(str) {
  return parseInt(str, 2);
}
```

106. Use the Conditional (Ternary) Operator

Description

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;` The following function uses an if-else statement to check a condition:

```
function findGreater(a, b) {
  if(a > b) {
    return "a is greater";
  }
  else {
```

```
    return "b is greater";
  }
}
```

This can be re-written using the `conditional operator`:

```
function findGreater(a, b) {
  return a > b ? "a is greater" : "b is greater";
}
```

Instructions

Use the `conditional operator` in the `checkEqual` function to check if two numbers are equal or not. The function should return either "Equal" or "Not Equal".

Challenge Seed

```
function checkEqual(a, b) {

}

checkEqual(1, 2);
```

Solution

```
function checkEqual(a, b) {
  return a === b ? "Equal" : "Not Equal";
}
```

107. Use Multiple Conditional (Ternary) Operators

Description

In the previous challenge, you used a single `conditional operator`. You can also chain them together to check for multiple conditions. The following function uses if, else if, and else statements to check multiple conditions:

```
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";
}
```

Instructions

Use multiple `conditional operators` in the `checkSign` function to check if a number is positive, negative or zero.

Challenge Seed

```
function checkSign(num) {

}

checkSign(10);
```

Solution

```
function checkSign(num) {
  return (num > 0) ? 'positive' : (num < 0) ? 'negative' : 'zero';
}
```

ES6

1. Explore Differences Between the var and let Keywords

Description

One of the biggest problems with declaring variables with the `var` keyword is that you can overwrite variable declarations without an error.

```
var camper = 'James';
var camper = 'David';
console.log(camper);
// logs 'David'
```

As you can see in the code above, the `camper` variable is originally declared as `James` and then overridden to be `David`. 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. If you were to replace `var` with `let` in the variable declarations of the code above, the result would be an error.

```
let camper = 'James';
let camper = 'David'; // throws an error
```

This error can be seen in the console of your browser. 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:

```
"use strict";
x = 3.14; // throws an error because x is not declared
```

Instructions

Update the code so it only uses the `let` keyword.

Challenge Seed

```
var catName;
var quote;
function catTalk() {
  "use strict";

  catName = "Oliver";
  quote = catName + " says Meow!";

}
catTalk();
```

Solution

```
let catName;
let quote;
function catTalk() {
  'use strict';

  catName = 'Oliver';
  quote = catName + ' says Meow!';
}
catTalk();
```

2. Compare Scopes of the var and let Keywords

Description

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. For example:

```
var numArray = [];
for (var i = 0; i < 3; i++) {
  numArray.push(i);
}
console.log(numArray);
// returns [0, 1, 2]
console.log(i);
// returns 3
```

With the `var` keyword, `i` is declared globally. So when `i++` is executed, it updates the global variable. This code is similar to the following:

```
var numArray = [];
var i;
for (i = 0; i < 3; i++) {
  numArray.push(i);
}
console.log(numArray);
// returns [0, 1, 2]
console.log(i);
// returns 3
```

This behavior will cause problems if you were to create a function and store it for later use inside a for loop that uses the `i` variable. This is because the stored function will always refer to the value of the updated global `i` variable.

```
var printNumTwo;
for (var i = 0; i < 3; i++) {
  if(i === 2){
    printNumTwo = function() {
      return i;
    };
  }
}
console.log(printNumTwo());
// returns 3
```

As you can see, `printNumTwo()` prints 3 and not 2. This is because the value assigned to `i` was updated and the `printNumTwo()` returns the global `i` and not the value `i` had when the function was created in the for loop. The `let` keyword does not follow this behavior:

```
'use strict';
let printNumTwo;
for (let i = 0; i < 3; i++) {
  if (i === 2) {
```

```
    printNumTwo = function() {
      return i;
    };
  }
}
console.log(printNumTwo());
// returns 2
console.log(i);
// returns "i is not defined"
```

`i` is not defined because it was not declared in the global scope. It is only declared within the for loop statement. `printNumTwo()` returned the correct value because three different `i` variables with unique values (0, 1, and 2) were created by the `let` keyword within the loop statement.

Instructions

Fix the code so that `i` declared in the if statement is a separate variable than `i` declared in the first line of the function. Be certain not to use the `var` keyword anywhere in your code. This exercise is designed to illustrate the difference between how `var` and `let` keywords assign scope to the declared variable. When programming a function similar to the one used in this exercise, it is often better to use different variable names to avoid confusion.

Challenge Seed

```
function checkScope() {
  'use strict';
  var i = 'function scope';
  if (true) {
    i = 'block scope';
    console.log('Block scope i is: ', i);
  }
  console.log('Function scope i is: ', i);
  return i;
}
```

Solution

```
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;
}
```

3. Declare a Read-Only Variable with the const Keyword

Description

The keyword `let` is not the only new way to declare variables. In ES6, you can also declare variables using 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.

```
"use strict"
const FAV_PET = "Cats";
FAV_PET = "Dogs"; // returns error
```

As you can see, trying to reassign a variable declared with `const` will throw an error. You should always name variables you don't want to reassign using the `const` keyword. This helps when you accidentally attempt to reassign a variable that is meant to stay constant. A common practice when naming constants is to use all uppercase letters, with words separated by an underscore.
Note: It is common for developers to use uppercase variable identifiers for immutable values and lowercase or camelCase for mutable values (objects and arrays). In a later challenge you will see an example of a lowercase variable identifier being used for an array.

Instructions

Change the code so that all variables are declared using `let` or `const`. Use `let` when you want the variable to change, and `const` when you want the variable to remain constant. Also, rename variables declared with `const` to conform to common practices, meaning constants should be in all caps.

Challenge Seed

```
function printManyTimes(str) {
  "use strict";

  // change code below this line

  var sentence = str + " is cool!";
  for(var i = 0; i < str.length; i+=2) {
    console.log(sentence);
  }

  // change code above this line

}
printManyTimes("freeCodeCamp");
```

Solution

```
function printManyTimes(str) {
  "use strict";

  // change code below this line

  const SENTENCE = str + " is cool!";
  for(let i = 0; i < str.length; i+=2) {
    console.log(SENTENCE);
  }

  // change code above this line

}
printManyTimes("freeCodeCamp");
```

4. Mutate an Array Declared with const

Description

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`. However, it is important to understand that objects (including arrays and functions) assigned to a variable using `const` are still mutable. Using the `const` declaration only prevents reassignment of the variable identifier.

```
"use strict";
const s = [5, 6, 7];
s = [1, 2, 3]; // throws error, trying to assign a const
s[2] = 45; // works just as it would with an array declared with var or let
console.log(s); // returns [5, 6, 45]
```

As you can see, you can mutate the object `[5, 6, 7]` itself and the variable `s` will still point to the altered array `[5, 6, 45]`. Like all arrays, the array elements in `s` are mutable, but because `const` was used, you cannot use the variable identifier `s` to point to a different array using the assignment operator.

Instructions

An array is declared as `const s = [5, 7, 2]`. Change the array to `[2, 5, 7]` using various element assignment.

Challenge Seed

```
const s = [5, 7, 2];
function editInPlace() {
  'use strict';
  // change code below this line

  // s = [2, 5, 7]; <- this is invalid

  // change code above this line
}
editInPlace();
```

Solution

```
const s = [5, 7, 2];
function editInPlace() {
  'use strict';
  // change code below this line

  // s = [2, 5, 7]; <- this is invalid
  s[0] = 2;
  s[1] = 5;
  s[2] = 7;
  // change code above this line
}
editInPlace();
```

5. Prevent Object Mutation

Description

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
obj.newProp = "Test"; // will be ignored. Mutation not allowed
console.log(obj);
// { name: "FreeCodeCamp", review:"Awesome"}
```

Instructions

In this challenge you are going to use `Object.freeze` to prevent mathematical constants from changing. You need to freeze the `MATH_CONSTANTS` object so that no one is able to alter the value of `PI`, add, or delete properties.

Challenge Seed

```
function freezeObj() {
  'use strict';
  const MATH_CONSTANTS = {
    PI: 3.14
  };
  // change code below this line


  // change code above this line
  try {
    MATH_CONSTANTS.PI = 99;
  } catch( ex ) {
    console.log(ex);
  }
  return MATH_CONSTANTS.PI;
}
const PI = freezeObj();
```

Solution

```
function freezeObj() {
  'use strict';
  const MATH_CONSTANTS = {
    PI: 3.14
  };
  // change code below this line
  Object.freeze(MATH_CONSTANTS);

  // change code above this line
  try {
    MATH_CONSTANTS.PI = 99;
  } catch( ex ) {
    console.log(ex);
  }
  return MATH_CONSTANTS.PI;
}
const PI = freezeObj();
```

6. Use Arrow Functions to Write Concise Anonymous Functions

Description

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;
}
```

ES6 provides us with the syntactic sugar to not have to write anonymous functions this way. Instead, you can use **arrow function syntax**:

```
const myFunc = () => {
  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.

Instructions

Rewrite the function assigned to the variable `magic` which returns a new `Date()` to use arrow function syntax. Also make sure nothing is defined using the keyword `var`.

Challenge Seed

```
var magic = function() {
  "use strict";
  return new Date();
};
```

Solution

```
const magic = () => {
  "use strict";
  return new Date();
};
```

7. Write Arrow Functions with Parameters

Description

Just like a regular function, you can pass arguments into an arrow function.

> // doubles input value and returns it
> const doubler = (item) => item * 2;

If an arrow function has a single argument, the parentheses enclosing the argument may be omitted.

> // the same function, without the argument parentheses
> const doubler = item => item * 2;

It is possible to pass more than one argument into an arrow function.

> // multiplies the first input value by the second and returns it
> const multiplier = (item, multi) => item * multi;

Instructions

Rewrite the `myConcat` function which appends contents of `arr2` to `arr1` so that the function uses arrow function syntax.

Challenge Seed

```
var myConcat = function(arr1, arr2) {
  "use strict";
  return arr1.concat(arr2);
};
// test your code
console.log(myConcat([1, 2], [3, 4, 5]));
```

Solution

```
const myConcat = (arr1, arr2) =>  {
  "use strict";
```

```
  return arr1.concat(arr2);
};
// test your code
console.log(myConcat([1, 2], [3, 4, 5]));
```

8. Write Higher Order Arrow Functions

Description

It's time to look at higher-order functions and their common pair, arrow functions. Arrow functions work really well when combined with higher-order functions, such as `map()`, `filter()`, and `reduce()`.
But what are these functions? Lets look at the simplest example `forEach()`, and run it on the following array of sample Facebook posts.

> let FBPosts = [
> {thumbnail: "someIcon", likes:432, shares: 600},
> {thumbnail: "Another icon", likes:300, shares: 501},
> {thumbnail: "Yet another", likes:40, shares: 550},
> {thumbnail: null, likes: 101, shares:0},
>]

Of the two `forEach()` versions below, both perform the exact same log function, and each takes an anonymous callback with a parameter `post`. The difference is the syntax. One uses an arrow function and the other does not.

> **ES5**
> FBpost.forEach(function(post) {
> console.log(post) // log each post here
> });
> **ES6**
> FBpost.forEach((post) => {
> console.log(post) // log each post here
> });

`filter()` is very similar. Below it will iterate over the `FBPosts` array, perform the logic to filter out the items that do not meet the requirements, and return a new array, `results`.

> let results = arr1.filter((post) => { return post.thumbnail !== null && post.likes > 100 && post.shares > 500 });
>
> console.log(results); // [{thumbnail: "someIcon", likes: 432, shares: 600}, {thumbnail: "Another icon", likes: 300, shares: 501}]

Instructions

Use arrow function syntax to compute the square of *only* the positive integers (decimal numbers are not integers) in the array `realNumberArray` and store the new array in the variable `squaredIntegers`.

Challenge Seed

```
const realNumberArray = [4, 5.6, -9.8, 3.14, 42, 6, 8.34, -2];
const squareList = (arr) => {
  "use strict";
  const positiveIntegers = arr.filter((num) => {
     // add code here
  });
  const squaredIntegers = positiveIntegers.map((num) => {
       // add code here
  });

  return squaredIntegers;
};
// test your code
const squaredIntegers = squareList(realNumberArray);
console.log(squaredIntegers);
```

Solution

```
const realNumberArray = [4, 5.6, -9.8, 3.14, 42, 6, 8.34, -2];
const squareList = (arr) => {
  "use strict";
  const positiveIntegers = arr.filter((num) => {
    return num >= 0 && Number.isInteger(num);
      // add code here
  });
  const squaredIntegers = positiveIntegers.map((num) => {
    // add code here
    return num ** 2;
  });
  // add code here
  return squaredIntegers;
};
// test your code
const squaredIntegers = squareList(realNumberArray);
```

9. Set Default Parameters for Your Functions

Description

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 `name` will 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.

Instructions

Modify the function `increment` by adding default parameters so that it will add 1 to `number` if `value` is not specified.

Challenge Seed

```
const increment = (function() {
  "use strict";
  return function increment(number, value) {
    return number + value;
  };
})();
console.log(increment(5, 2)); // returns 7
console.log(increment(5)); // returns 6
```

Solution

```
// solution required
```

10. Use the Rest Operator with Function Parameters

Description

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. 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.

Instructions

Modify the function `sum` using the rest parameter in such a way that the function `sum` is able to take any number of arguments and return their sum.

Challenge Seed

```
const sum = (function() {
  "use strict";
  return function sum(x, y, z) {
    const args = [ x, y, z ];
    return args.reduce((a, b) => a + b, 0);
  };
})();
console.log(sum(1, 2, 3)); // 6
```

Solution

```
// solution required
```

11. Use the Spread Operator to Evaluate Arrays In-Place

Description

ES6 introduces the spread operator, which allows us to expand arrays and other expressions in places where multiple parameters or elements are expected. The ES5 code below uses `apply()` to compute the maximum value in an array:

```
var arr = [6, 89, 3, 45];
var maximus = Math.max.apply(null, arr); // returns 89
```

We had to use `Math.max.apply(null, arr)` because `Math.max(arr)` returns `NaN`. `Math.max()` expects comma-separated arguments, but not an array. The spread operator makes this syntax much better to read and maintain.

```
const arr = [6, 89, 3, 45];
const maximus = Math.max(...arr); // returns 89
```

`...arr` returns an unpacked array. In other words, it *spreads* the array. However, the spread operator only works in-place, like in an argument to a function or in an array literal. The following code will not work:

```
const spreaded = ...arr; // will throw a syntax error
```

Instructions

Copy all contents of `arr1` into another array `arr2` using the spread operator.

Challenge Seed

```
const arr1 = ['JAN', 'FEB', 'MAR', 'APR', 'MAY'];
let arr2;
(function() {
  "use strict";
  arr2 = []; // change this line
})();
console.log(arr2);
```

Solution

```
const arr1 = ['JAN', 'FEB', 'MAR', 'APR', 'MAY'];
let arr2;
(function() {
  "use strict";
  arr2 = [...arr1]; // change this line
})();
console.log(arr2);
```

12. Use Destructuring Assignment to Assign Variables from Objects

Description

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.x` into `a`, `voxel.y` into `b`, and `voxel.z` into `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.

Instructions

Use destructuring to obtain the average temperature for tomorrow from the input object `AVG_TEMPERATURES`, and assign value with key `tomorrow` to `tempOfTomorrow` in line.

Challenge Seed

```
const AVG_TEMPERATURES = {
  today: 77.5,
  tomorrow: 79
};

function getTempOfTmrw(avgTemperatures) {
  "use strict";
  // change code below this line
  const tempOfTomorrow = undefined; // change this line
  // change code above this line
  return tempOfTomorrow;
}

console.log(getTempOfTmrw(AVG_TEMPERATURES)); // should be 79
```

Solution

```
const AVG_TEMPERATURES = {
  today: 77.5,
  tomorrow: 79
};

function getTempOfTmrw(avgTemperatures) {
  "use strict";
  // change code below this line
  const {tomorrow:tempOfTomorrow} = avgTemperatures; // change this line
  // change code above this line
  return tempOfTomorrow;
}

console.log(getTempOfTmrw(AVG_TEMPERATURES)); // should be 79
```

13. Use Destructuring Assignment to Assign Variables from Nested Objects

Description

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 `startX` is assigned the value of `a.start.x`.

Instructions

Use destructuring assignment to obtain `max` of `forecast.tomorrow` and assign it to `maxOfTomorrow`.

Challenge Seed

```
const LOCAL_FORECAST = {
  today: { min: 72, max: 83 },
  tomorrow: { min: 73.3, max: 84.6 }
};

function getMaxOfTmrw(forecast) {
  "use strict";
  // change code below this line
  const maxOfTomorrow = undefined; // change this line
  // change code above this line
  return maxOfTomorrow;
}

console.log(getMaxOfTmrw(LOCAL_FORECAST)); // should be 84.6
```

Solution

```
const LOCAL_FORECAST = {
  today: { min: 72, max: 83 },
  tomorrow: { min: 73.3, max: 84.6 }
};
```

```
function getMaxOfTmrw(forecast) {
  "use strict";
  // change code below this line
   const {tomorrow : {max : maxOfTomorrow}} = forecast; // change this line
  // change code above this line
  return maxOfTomorrow;
}

console.log(getMaxOfTmrw(LOCAL_FORECAST)); // should be 84.6
```

14. Use Destructuring Assignment to Assign Variables from Arrays

Description

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

The variable `a` is assigned the first value of the array, and `b` is assigned the second value of the array. We can also access the value at any index in an array with destructuring by using commas to reach the desired index:

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

Instructions

Use destructuring assignment to swap the values of `a` and `b` so that `a` receives the value stored in `b`, and `b` receives the value stored in `a`.

Challenge Seed

```
let a = 8, b = 6;
(() => {
  "use strict";
  // change code below this line

  // change code above this line
})();
console.log(a); // should be 6
console.log(b); // should be 8
```

Solution

```
// solution required
```

15. Use Destructuring Assignment with the Rest Operator to Reassign Array Elements

Description

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 `a` and `b` take 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. The rest element only works correctly as the last variable in the list. As in, you cannot use the rest operator to catch a subarray that leaves out last element of the original array.

Instructions

Use destructuring assignment with the rest operator to perform an effective `Array.prototype.slice()` so that `arr` is a sub-array of the original array `source` with the first two elements omitted.

Challenge Seed

```
const source = [1,2,3,4,5,6,7,8,9,10];
function removeFirstTwo(list) {
  "use strict";
  // change code below this line
  const arr = list; // change this
  // change code above this line
  return arr;
}
const arr = removeFirstTwo(source);
console.log(arr); // should be [3,4,5,6,7,8,9,10]
console.log(source); // should be [1,2,3,4,5,6,7,8,9,10];
```

Solution

```
const source = [1,2,3,4,5,6,7,8,9,10];
function removeFirstTwo(list) {
  "use strict";
  // change code below this line
  const [, , ...arr] = list;
  // change code above this line
  return arr;
}
const arr = removeFirstTwo(source);
```

16. Use Destructuring Assignment to Pass an Object as a Function's Parameters

Description

In some cases, you can destructure the object in a function argument itself. Consider the code below:

```
const profileUpdate = (profileData) => {
  const { name, age, nationality, location } = profileData;
  // do something with these variables
}
```

This effectively destructures the object sent into the function. This can also be done in-place:

```
const profileUpdate = ({ name, age, nationality, location }) => {
  /* do something with these fields */
}
```

This removes some extra lines and makes our code look neat. This has the added benefit of not having to manipulate an entire object in a function; only the fields that are needed are copied inside the function.

Instructions

Use destructuring assignment within the argument to the function `half` to send only `max` and `min` inside the function.

Challenge Seed

```
const stats = {
  max: 56.78,
  standard_deviation: 4.34,
  median: 34.54,
  mode: 23.87,
  min: -0.75,
  average: 35.85
};
const half = (function() {
  "use strict"; // do not change this line

  // change code below this line
  return function half(stats) {
    // use function argument destructuring
    return (stats.max + stats.min) / 2.0;
  };
  // change code above this line

})();
console.log(stats); // should be object
console.log(half(stats)); // should be 28.015
```

Solution

```
// solution required
```

17. Create Strings using Template Literals

Description

A new feature of ES6 is the template literal. This is a special type of string that makes creating complex strings easier. Template literals allow you to create multi-line strings and to use string interpolation features to create strings. Consider the code below:

```
const person = {
  name: "Zodiac Hasbro",
  age: 56
};

// Template literal with multi-line and string interpolation
const greeting = `Hello, my name is ${person.name}!
I am ${person.age} years old.`;

console.log(greeting); // prints
// Hello, my name is Zodiac Hasbro!
// I am 56 years old.
```

A lot of things happened there. Firstly, the example uses backticks (`` ` ``), not quotes (`'` or `"`), to wrap the string. Secondly, notice that the string is multi-line, both in the code and the output. This saves inserting `\n` within strings. The `${variable}` syntax used above is a placeholder. Basically, you won't have to use concatenation with the `+` operator anymore. To add variables to strings, you just drop the variable in a template string and wrap it with `${` and `}` . Similarly, you can include other expressions in your string literal, for example `${a + b}` . This new way of creating strings gives you more flexibility to create robust strings.

Instructions

Use template literal syntax with backticks to display each entry of the `result` object's `failure` array. Each entry should be wrapped inside an `li` element with the class attribute `text-warning`, and listed within the `resultDisplayArray`. Use an iterator method (any kind of loop) to get the desired output.

Challenge Seed

```
const result = {
  success: ["max-length", "no-amd", "prefer-arrow-functions"],
  failure: ["no-var", "var-on-top", "linebreak"],
  skipped: ["id-blacklist", "no-dup-keys"]
};
function makeList(arr) {
  "use strict";

  // change code below this line
  const resultDisplayArray = null;
  // change code above this line

  return resultDisplayArray;
}
/**
 * makeList(result.failure) should return:
 * [ `<li class="text-warning">no-var</li>`,
 *   `<li class="text-warning">var-on-top</li>`,
 *   `<li class="text-warning">linebreak</li>` ]
 **/
const resultDisplayArray = makeList(result.failure);
```

Solution

```
const result = {
  success: ["max-length", "no-amd", "prefer-arrow-functions"],
  failure: ["no-var", "var-on-top", "linebreak"],
  skipped: ["id-blacklist", "no-dup-keys"]
};
function makeList(arr) {
  "use strict";

  const resultDisplayArray = arr.map(val => `<li class="text-warning">${val}</li>`);

  return resultDisplayArray;
}
/**
 * makeList(result.failure) should return:
 * [ `<li class="text-warning">no-var</li>`,
 *   `<li class="text-warning">var-on-top</li>`,
 *   `<li class="text-warning">linebreak</li>` ]
 **/
const resultDisplayArray = makeList(result.failure);
```

18. Write Concise Object Literal Declarations Using Simple Fields

Description

ES6 adds some nice support for easily defining object literals. Consider the following code:

```
const getMousePosition = (x, y) => ({
  x: x,
  y: y
});
```

`getMousePosition` is a simple function that returns an object containing two fields. ES6 provides the syntactic sugar to eliminate the redundancy of having to write `x: x`. You can simply write `x` once, and it will be converted to `x: x` (or something equivalent) under the hood. Here is the same function from above rewritten to use this new syntax:

```
const getMousePosition = (x, y) => ({ x, y });
```

Instructions

Use simple fields with object literals to create and return a `Person` object with `name`, `age` and `gender` properties.

Challenge Seed

```
const createPerson = (name, age, gender) => {
  "use strict";
  // change code below this line
  return {
    name: name,
    age: age,
    gender: gender
  };
  // change code above this line
};
console.log(createPerson("Zodiac Hasbro", 56, "male")); // returns a proper object
```

Solution

```
const createPerson = (name, age, gender) => {
  "use strict";
  return {
    name,
    age,
    gender
  };
};
```

19. Write Concise Declarative Functions with ES6

Description

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

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

With ES6, You can remove the `function` keyword 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}.`;
  }
};
```

Instructions

Refactor the function `setGear` inside the object `bicycle` to use the shorthand syntax described above.

Challenge Seed

```
// change code below this line
const bicycle = {
  gear: 2,
  setGear: function(newGear) {
    this.gear = newGear;
  }
};
// change code above this line
bicycle.setGear(3);
console.log(bicycle.gear);
```

Solution

```
const bicycle = {
  gear: 2,
  setGear(newGear) {
    this.gear = newGear;
  }
};
bicycle.setGear(3);
```

20. Use class Syntax to Define a Constructor Function

Description

ES6 provides a new syntax to help create objects, using the keyword class. This is to be noted, that the `class` syntax is just a syntax, and not a full-fledged class based implementation of object oriented paradigm, unlike in languages like Java, or Python, or Ruby etc. In ES5, we usually define a constructor function, and use the `new` keyword to instantiate an object.

```
var SpaceShuttle = function(targetPlanet){
  this.targetPlanet = targetPlanet;
}
var zeus = new SpaceShuttle('Jupiter');
```

The class syntax simply replaces the constructor function creation:

```
class SpaceShuttle {
  constructor(targetPlanet){
    this.targetPlanet = targetPlanet;
  }
}
const zeus = new SpaceShuttle('Jupiter');
```

Notice that the `class` keyword declares a new function, and a constructor was added, which would be invoked when `new` is called - to create a new object.

Note

UpperCamelCase should be used by convention for ES6 class names, as in `SpaceShuttle` used above.

Instructions

Use `class` keyword and write a proper constructor to create the `Vegetable` class. The `Vegetable` lets you create a vegetable object, with a property `name` , to be passed to constructor.

Challenge Seed

```
function makeClass() {
  "use strict";
  /* Alter code below this line */
```

```
  /* Alter code above this line */
  return Vegetable;
}
const Vegetable = makeClass();
const carrot = new Vegetable('carrot');
console.log(carrot.name); // => should be 'carrot'
```

Solution

```
function makeClass() {
  "use strict";
  /* Alter code below this line */
  class Vegetable {
    constructor(name){
      this.name = name;
    }
  }
  /* Alter code above this line */
  return Vegetable;
}
const Vegetable = makeClass();
const carrot = new Vegetable('carrot');
console.log(carrot.name); // => should be 'carrot'
```

21. Use getters and setters to Control Access to an Object

Description

You can obtain values from an object, and set a value of a property within an object. These are classically called getters and setters. Getter functions are meant to simply return (get) the value of an object's private variable to the user without the user directly accessing the private variable. Setter functions are meant to modify (set) the value of an object's private variable based on the value passed into the setter function. This change could involve calculations, or even overwriting the previous value completely.

```
class Book {
  constructor(author) {
    this._author = author;
  }
  // getter
  get writer(){
    return this._author;
  }
  // setter
  set writer(updatedAuthor){
    this._author = updatedAuthor;
  }
}
const lol = new Book('anonymous');
console.log(lol.writer);  // anonymous
lol.writer = 'wut';
console.log(lol.writer);  // wut
```

Notice the syntax we are using to invoke the getter and setter - as if they are not even functions. Getters and setters are important, because they hide internal implementation details.

Note:
It is a convention to precede the name of a private variable with an underscore (_). The practice itself does not make a variable private.

Instructions

Use `class` keyword to create a Thermostat class. The constructor accepts Fahrenheit temperature. Now create `getter` and `setter` in the class, to obtain the temperature in Celsius scale. Remember that `C = 5/9 * (F - 32)` and `F = C * 9.0 / 5 + 32`, where F is the value of temperature in Fahrenheit scale, and C is the value of the same temperature in Celsius scale Note When you implement this, you would be tracking the temperature inside the class in one scale - either Fahrenheit or Celsius. This is the power of getter or setter - you are creating an API for another user, who would get the correct result, no matter which one you track. In other words, you are abstracting implementation details from the consumer.

Challenge Seed

```
function makeClass() {
  "use strict";
  /* Alter code below this line */

  /* Alter code above this line */
  return Thermostat;
}
const Thermostat = makeClass();
const thermos = new Thermostat(76); // setting in Fahrenheit scale
let temp = thermos.temperature; // 24.44 in C
thermos.temperature = 26;
temp = thermos.temperature; // 26 in C
```

Solution

```
function makeClass() {
  "use strict";
  /* Alter code below this line */
  class Thermostat {
    constructor(fahrenheit) {
      this._tempInCelsius = 5/9 * (fahrenheit - 32);
    }
    get tempInCelsius(){
      return _tempInCelsius;
    }
    set tempInCelsius(newTemp){
      this._tempInCelsius = newTemp;
    }
  }
  /* Alter code above this line */
  return Thermostat;
}
const Thermostat = makeClass();
const thermos = new Thermostat(76); // setting in Fahrenheit scale
let temp = thermos.temperature; // 24.44 in C
thermos.temperature = 26;
temp = thermos.temperature; // 26 in C
```

22. Understand the Differences Between import and require

Description

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_functions` has 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 `import` syntax, 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 `import` statement. **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_modules` directory first trying to load it as a dependency.

Instructions

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

Challenge Seed

```
"use strict";
capitalizeString("hello!");
```

Before Test

```
self.require = function (str) {
  if (str === 'string_functions') {
    return {
      capitalizeString: str => str.toUpperCase()
    }
  }
};
```

Solution

```
import { capitalizeString } from 'string_functions';
capitalizeString("hello!");
```

23. Use export to Reuse a Code Block

Description

In the previous challenge, you learned about `import` and 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`, `export` is 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 `import` syntax 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.

Alternatively, if you would like to compact all your `export` statements into one line, you can take this approach:

```
const capitalizeString = (string) => {
  return string.charAt(0).toUpperCase() + string.slice(1);
}
const foo = "bar";
export { capitalizeString, foo }
```

Either approach is perfectly acceptable.

Instructions

Below are two variables that I want to make available for other files to use. Utilizing the first way I demonstrated `export`, export the two variables.

Challenge Seed

```
"use strict";
const foo = "bar";
const bar = "foo";
```

Before Test

```
self.exports = function(){};
```

Solution

```
"use strict";
export const foo = "bar";
export const bar = "foo";
```

24. Use * to Import Everything from a File

Description

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.

Instructions

The code below requires the contents of a file, `"capitalize_strings"`, found in the same directory as it, imported. Add the appropriate `import *` statement to the top of the file, using the object provided.

Challenge Seed

```
"use strict";
```

Before Test

```
self.require = function(str) {
  if (str === 'capitalize_strings') {
    return {
      capitalize: str => str.toUpperCase(),
      lowercase: str => str.toLowerCase()
    }
  }
};
```

Solution

```
import * as capitalize_strings from "capitalize_strings";
```

25. Create an Export Fallback with export default

Description

In the `export` lesson, 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 `export` syntax 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 default` is 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`

Instructions

The following function should be the fallback value for the module. Please add the necessary code to do so.

Challenge Seed

```
"use strict";
function subtract(x,y) {return x - y;}
```

Before Test

```
self.exports = function(){};
```

Solution

```
export default function subtract(x,y) {return x - y;}
```

26. Import a Default Export

Description

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

Instructions

In the following code, please import the default export, `subtract`, from the file `"math_functions"`, found in the same directory as this file.

Challenge Seed

```
"use strict";
subtract(7,4);
```

Before Test

```
self.require = function(str) {
  if (str === 'math_functions') {
    return function(a, b) {
      return a - b;
    }
  }
};
```

Solution

```
import subtract from "math_functions";
subtract(7,4);
```

Regular Expressions

1. Using the Test Method

Description

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);
// Returns true
```

Instructions

Apply the regex `myRegex` on the string `myString` using the `.test()` method.

Challenge Seed

```
let myString = "Hello, World!";
let myRegex = /Hello/;
let result = myRegex; // Change this line
```

Solution

```
// solution required
```

2. Match Literal Strings

Description

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);
> // Returns 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);
> // Returns false

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

Instructions

Complete the regex `waldoRegex` to find `"Waldo"` in the string `waldoIsHiding` with a literal match.

Challenge Seed

```
let waldoIsHiding = "Somewhere Waldo is hiding in this text.";
let waldoRegex = /search/; // Change this line
let result = waldoRegex.test(waldoIsHiding);
```

Solution

```
// solution required
```

3. Match a Literal String with Different Possibilities

Description

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 `"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/`.

Instructions

Complete the regex `petRegex` to match the pets `"dog"`, `"cat"`, `"bird"`, or `"fish"`.

Challenge Seed

```
let petString = "James has a pet cat.";
let petRegex = /change/; // Change this line
let result = petRegex.test(petString);
```

Solution

```
// solution required
```

4. Ignore Case While Matching

Description

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"`.

Instructions

Write a regex `fccRegex` to match `"freeCodeCamp"`, no matter its case. Your regex should not match any abbreviations or variations with spaces.

Challenge Seed

```
let myString = "freeCodeCamp";
let fccRegex = /change/; // Change this line
let result = fccRegex.test(myString);
```

Solution

```
// solution required
```

5. Extract Matches

Description

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"]
```

Instructions

Apply the `.match()` method to extract the word `coding`.

Challenge Seed

```
let extractStr = "Extract the word 'coding' from this string.";
let codingRegex = /change/; // Change this line
let result = extractStr; // Change this line
```

Solution

```
// solution required
```

6. Find More Than the First Match

Description

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);
// Returns ["Repeat"]
```

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

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

Instructions

Using the regex `starRegex`, find and extract both `"Twinkle"` words from the string `twinkleStar`. **Note**
You can have multiple flags on your regex like `/search/gi`

Challenge Seed

```
let twinkleStar = "Twinkle, twinkle, little star";
let starRegex = /change/; // Change this line
let result = twinkleStar; // Change this line
```

Solution

```
let twinkleStar = "Twinkle, twinkle, little star";
let starRegex = /twinkle/gi;
let result = twinkleStar.match(starRegex);
```

7. Match Anything with Wildcard Period

Description

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./;
humStr.match(huRegex); // Returns ["hum"]
hugStr.match(huRegex); // Returns ["hug"]
```

Instructions

Complete the regex `unRegex` so that it matches the strings `"run"`, `"sun"`, `"fun"`, `"pun"`, `"nun"`, and `"bun"`. Your regex should use the wildcard character.

Challenge Seed

```
let exampleStr = "Let's have fun with regular expressions!";
let unRegex = /change/; // Change this line
let result = unRegex.test(exampleStr);
```

Solution

```
// solution required
```

8. Match Single Character with Multiple Possibilities

Description

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); // Returns ["big"]
bagStr.match(bgRegex); // Returns ["bag"]
bugStr.match(bgRegex); // Returns ["bug"]
bogStr.match(bgRegex); // Returns null
```

Instructions

Use a character class with vowels (`a`, `e`, `i`, `o`, `u`) in your regex `vowelRegex` to find all the vowels in the string `quoteSample`. **Note**

Be sure to match both upper- and lowercase vowels.

Challenge Seed

```
let quoteSample = "Beware of bugs in the above code; I have only proved it correct, not tried it.";
let vowelRegex = /change/; // Change this line
let result = vowelRegex; // Change this line
```

Solution

```
// solution required
```

9. Match Letters of the Alphabet

Description

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); // Returns ["cat"]
batStr.match(bgRegex); // Returns ["bat"]
matStr.match(bgRegex); // Returns null
```

Instructions

Match all the letters in the string `quoteSample`. **Note**
Be sure to match both upper- and lowercase **letters.**

Challenge Seed

```
let quoteSample = "The quick brown fox jumps over the lazy dog.";
let alphabetRegex = /change/; // Change this line
let result = alphabetRegex; // Change this line
```

Solution

```
// solution required
```

10. Match Numbers and Letters of the Alphabet

Description

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;
// matches all letters and numbers in jennyStr
jennyStr.match(myRegex);
```

Instructions

Create a single regex that matches a range of letters between `h` and `s`, and a range of numbers between `2` and `6`. Remember to include the appropriate flags in the regex.

Challenge Seed

```
let quoteSample = "Blueberry 3.141592653s are delicious.";
let myRegex = /change/; // Change this line
let result = myRegex; // Change this line
```

Solution

```
// solution required
```

11. Match Single Characters Not Specified

Description

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.

Instructions

Create a single regex that matches all characters that are not a number or a vowel. Remember to include the appropriate flags in the regex.

Challenge Seed

```
let quoteSample = "3 blind mice.";
let myRegex = /change/; // Change this line
let result = myRegex; // Change this line
```

Solution

```
// solution required
```

12. Match Characters that Occur One or More Times

Description

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.

Instructions

You want to find matches when the letter `s` occurs one or more times in `"Mississippi"`. Write a regex that uses the `+` sign.

Challenge Seed

```
let difficultSpelling = "Mississippi";
let myRegex = /change/; // Change this line
let result = difficultSpelling.match(myRegex);
```

Solution

```
let difficultSpelling = "Mississippi";
let myRegex = /s+/g; // Change this line
let result = difficultSpelling.match(myRegex);
```

13. Match Characters that Occur Zero or More Times

Description

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); // Returns ["gooooooooo"]**
> **gPhrase.match(goRegex); // Returns ["g"]**
> **oPhrase.match(goRegex); // Returns null**

Instructions

Create a regex `chewieRegex` that uses the `*` character to match all the upper and lowercase `"a"` characters in `chewieQuote`. Your regex does not need flags, and it should not match any of the other quotes.

Challenge Seed

```
let chewieQuote = "Aaaaaaaaaaaaaaaarrrgh!";
let chewieRegex = /change/; // Change this line
let result = chewieQuote.match(chewieRegex);
```

Solution

```
let chewieQuote = "Aaaaaaaaaaaaaaaarrrgh!";
let chewieRegex = /Aa*/;
```

```
let result = chewieQuote.match(chewieRegex);
```

14. Find Characters with Lazy Matching

Description

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"]`.

Instructions

Fix the regex `/<.*>/` to return the HTML tag `<h1>` and not the text `"<h1>Winter is coming</h1>"`. Remember the wildcard `.` in a regular expression matches any character.

Challenge Seed

```
let text = "<h1>Winter is coming</h1>";
let myRegex = /<.*>/; // Change this line
let result = text.match(myRegex);
```

Solution

```
// solution required
```

15. Find One or More Criminals in a Hunt

Description

Time to pause and test your new regex writing skills. A group of criminals escaped from jail and ran away, but you don't know how many. However, you do know that they stay close together when they are around other people. You are responsible for finding all of the criminals at once. Here's an example to review how to do this: The regex `/z+/` matches the letter `z` when it appears one or more times in a row. It would find matches in all of the following strings:

> "z"
> "zzzzzz"
> "ABCzzzz"
> "zzzzABC"
> "abczzzzzzzzzzzzzzzzzzzzzabc"

But it does not find matches in the following strings since there are no letter `z` characters:

> ""
> "ABC"
> "abcabc"

Instructions

Write a `greedy` regex that finds one or more criminals within a group of other people. A criminal is represented by the capital letter `C`.

Challenge Seed

```
// example crowd gathering
let crowd = 'P1P2P3P4P5P6CCCP7P8P9';

let reCriminals = /./; // Change this line

let matchedCriminals = crowd.match(reCriminals);
console.log(matchedCriminals);
```

Solution

```
// solution required
```

16. Match Beginning String Patterns

Description

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);**
> **// Returns true**
> **let notFirst = "You can't find Ricky now.";**
> **firstRegex.test(notFirst);**
> **// Returns false**

Instructions

Use the `caret` character in a regex to find `"Cal"` only in the beginning of the string `rickyAndCal` .

Challenge Seed

```
let rickyAndCal = "Cal and Ricky both like racing.";
let calRegex = /change/; // Change this line
let result = calRegex.test(rickyAndCal);
```

Solution

```
// solution required
```

17. Match Ending String Patterns

Description

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);
// Returns true
let noEnding = "Sometimes a story will have to end";
storyRegex.test(noEnding);
// Returns false
```

Instructions

Use the anchor character (`$`) to match the string `"caboose"` at the end of the string `caboose` .

Challenge Seed

```
let caboose = "The last car on a train is the caboose";
let lastRegex = /change/; // Change this line
let result = lastRegex.test(caboose);
```

Solution

```
// solution required
```

18. Match All Letters and Numbers

Description

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); // Returns true
shortHand.test(numbers); // Returns true
longHand.test(varNames); // Returns true
shortHand.test(varNames); // Returns true
```

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

Instructions

Use the shorthand character class `\w` to count the number of alphanumeric characters in various quotes and strings.

Challenge Seed

```
let quoteSample = "The five boxing wizards jump quickly.";
let alphabetRegexV2 = /change/; // Change this line
let result = quoteSample.match(alphabetRegexV2).length;
```

Solution

```
// solution required
```

19. Match Everything But Letters and Numbers

Description

You've learned that you can use a shortcut to match alphanumerics `[A-Za-z0-9_]` using `\w`. A natural pattern you might want to search for is the opposite of alphanumerics. You can search for the opposite of the `\w` with `\W`. Note, the opposite pattern uses a capital letter. This shortcut is the same as `[^A-Za-z0-9_]`.

```
let shortHand = /\W/;
let numbers = "42%";
let sentence = "Coding!";
numbers.match(shortHand); // Returns ["%"]
sentence.match(shortHand); // Returns ["!"]
```

Instructions

Use the shorthand character class `\W` to count the number of non-alphanumeric characters in various quotes and strings.

Challenge Seed

```
let quoteSample = "The five boxing wizards jump quickly.";
let nonAlphabetRegex = /change/; // Change this line
let result = quoteSample.match(nonAlphabetRegex).length;
```

Solution

```
// solution required
```

20. Match All Numbers

Description

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.

Instructions

Use the shorthand character class `\d` to count how many digits are in movie titles. Written out numbers ("six" instead of 6) do not count.

Challenge Seed

```
let numString = "Your sandwich will be $5.00";
let numRegex = /change/; // Change this line
let result = numString.match(numRegex).length;
```

Solution

```
// solution required
```

21. Match All Non-Numbers

Description

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.

Instructions

Use the shorthand character class for non-digits `\D` to count how many non-digits are in movie titles.

Challenge Seed

```
let numString = "Your sandwich will be $5.00";
let noNumRegex = /change/; // Change this line
let result = numString.match(noNumRegex).length;
```

Solution

```
// solution required
```

22. Restrict Possible Usernames

Description

Usernames are used everywhere on the internet. They are what give users a unique identity on their favorite sites. You need to check all the usernames in a database. Here are some simple rules that users have to follow when creating their username. 1) The only numbers in the username have to be at the end. There can be zero or more of them at the end. 2) Username letters can be lowercase and uppercase. 3) Usernames have to be at least two characters long. A two-letter username can only use alphabet letter characters.

Instructions

Change the regex `userCheck` to fit the constraints listed above.

Challenge Seed

```
let username = "JackOfAllTrades";
let userCheck = /change/; // Change this line
let result = userCheck.test(username);
```

Solution

```
const userCheck = /^[A-Za-z]{2,}\d*$/;
```

23. Match Whitespace

Description

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);
// Returns [" ", " "]
```

Instructions

Change the regex `countWhiteSpace` to look for multiple whitespace characters in a string.

Challenge Seed

```
let sample = "Whitespace is important in separating words";
let countWhiteSpace = /change/; // Change this line
let result = sample.match(countWhiteSpace);
```

Solution

```
let sample = "Whitespace is important in separating words";
let countWhiteSpace = /\s/g;
let result = sample.match(countWhiteSpace);
```

24. Match Non-Whitespace Characters

Description

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; // Returns 32
```

Instructions

Change the regex `countNonWhiteSpace` to look for multiple non-whitespace characters in a string.

Challenge Seed

```
let sample = "Whitespace is important in separating words";
let countNonWhiteSpace = /change/; // Change this line
let result = sample.match(countNonWhiteSpace);
```

Solution

```
// solution required
```

25. Specify Upper and Lower Number of Matches

Description

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); // Returns true**
> **multipleA.test(A2); // Returns false**

Instructions

Change the regex `ohRegex` to match only `3` to `6` letter `h`'s in the word `"Oh no"`.

Challenge Seed

```
let ohStr = "Ohhh no";
let ohRegex = /change/; // Change this line
let result = ohRegex.test(ohStr);
```

Solution

```
// solution required
```

26. Specify Only the Lower Number of Matches

Description

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); // Returns true**
> **multipleA.test(A2); // Returns false**
> **multipleA.test(A100); // Returns true**

Instructions

Change the regex `haRegex` to match the word `"Hazzah"` only when it has four or more letter `z`'s.

Challenge Seed

```
let haStr = "Hazzzzah";
let haRegex = /change/; // Change this line
let result = haRegex.test(haStr);
```

Solution

```
// solution required
```

27. Specify Exact Number of Matches

Description

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); // Returns false
multipleHA.test(A3); // Returns true
multipleHA.test(A100); // Returns false
```

Instructions

Change the regex `timRegex` to match the word `"Timber"` only when it has four letter `m`'s.

Challenge Seed

```
let timStr = "Timmmmber";
let timRegex = /change/; // Change this line
let result = timRegex.test(timStr);
```

Solution

```
// solution required
```

28. Check for All or None

Description

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); // Returns true
rainbowRegex.test(british); // Returns true
```

Instructions

Change the regex `favRegex` to match both the American English (favorite) and the British English (favourite) version of the word.

Challenge Seed

```
let favWord = "favorite";
let favRegex = /change/; // Change this line
let result = favRegex.test(favWord);
```

Solution

```
let favWord = "favorite";
let favRegex = /favou?r/;
let result = favRegex.test(favWord);
```

29. Positive and Negative Lookahead

Description

`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); // Returns ["q"]
noquit.match(qRegex); // Returns ["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); // Returns true
```

Instructions

Use `lookaheads` in the `pwRegex` to match passwords that are greater than 5 characters long, do not begin with numbers, and have two consecutive digits.

Challenge Seed

```
let sampleWord = "astronaut";
let pwRegex = /change/; // Change this line
let result = pwRegex.test(sampleWord);
```

Solution

```
var pwRegex = /(?=\w{5})(?=\D*\d{2})/;
```

30. Reuse Patterns Using Capture Groups

Description

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); // 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.

Instructions

Use `capture groups` in `reRegex` to match numbers that are repeated only three times in a string, each separated by a space.

Challenge Seed

```
let repeatNum = "42 42 42";
let reRegex = /change/; // Change this line
let result = reRegex.test(repeatNum);
```

Solution

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

31. Use Capture Groups to Search and Replace

Description

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"

Instructions

Write a regex so that it will search for the string `"good"` . Then update the `replaceText` variable to replace `"good"` with `"okey-dokey"` .

Challenge Seed

```
let huhText = "This sandwich is good.";
let fixRegex = /change/; // Change this line
let replaceText = ""; // Change this line
let result = huhText.replace(fixRegex, replaceText);
```

Solution

```
// solution required
```

32. Remove Whitespace from Start and End

Description

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.

Instructions

Write a regex and use the appropriate string methods to remove whitespace at the beginning and end of strings.
Note
The `.trim()` method would work here, but you'll need to complete this challenge using regular expressions.

Challenge Seed

```
let hello = "   Hello, World!  ";
let wsRegex = /change/; // Change this line
let result = hello; // Change this line
```

Solution

```
let hello = "   Hello, World!  ";
let wsRegex = /^(\s+)(.+[^\s])(\s+)$/;
let result = hello.replace(wsRegex, '$2');
```

Debugging

1. Use the JavaScript Console to Check the Value of a Variable

Description

Both Chrome and Firefox have excellent JavaScript consoles, also known as DevTools, for debugging your JavaScript. You can find Developer tools in your Chrome's menu or Web Console in Firefox's menu. If you're using a different browser, or a mobile phone, we strongly recommend switching to desktop Firefox or Chrome. The `console.log()` method, which "prints" the output of what's within its parentheses to the console, will likely be the most helpful debugging tool. Placing it at strategic points in your code can show you the intermediate values of variables. It's good practice to have an idea of what the output should be before looking at what it is. Having check points to see the status of your calculations throughout your code will help narrow down where the problem is. Here's an example to print 'Hello world!' to the console: `console.log('Hello world!');`

Instructions

Use the `console.log()` method to print the value of the variable `a` where noted in the code.

Challenge Seed

```
let a = 5;
let b = 1;
a++;
// Add your code below this line


let sumAB = a + b;
console.log(sumAB);
```

Solution

```
var a = 5; console.log(a);
```

2. Understanding the Differences between the freeCodeCamp and Browser Console

Description

You may have noticed that some freeCodeCamp JavaScript challenges include their own console. This console behaves a little differently than the browser console you used in the last challenge. The following challenge is meant to highlight some of the differences between the freeCodeCamp console and the browser console. First, the browser console. When you load and run an ordinary JavaScript file in your browser the `console.log()` statements will print exactly what you tell them to print to the browser console the exact number of times you requested. In your in-browser text editor the process is slightly different and can be confusing at first. Values passed to `console.log()` in the text editor block run each set of tests as well as one more time for any function calls that you have in your code. This lends itself to some interesting behavior and might trip you up in the beginning, because a logged value that you expect to see only once may print out many more times depending on the number of tests and the values being passed to those tests. If you would like to see only your single output and not have to worry about running through the test cycles, you can use `console.clear()`.

Instructions

Use `console.log()` to print the variables in the code where indicated.

Challenge Seed

```
// Open your browser console
let outputTwo = "This will print to the browser console 2 times";
```

```
// Use console.log() to print the outputTwo variable


let outputOne = "Try to get this to log only once to the browser console";
// Use console.clear() in the next line to print the outputOne only once


// Use console.log() to print the outputOne variable
```

Solution

```
let outputTwo = "This will print to the browser console 2 times"; console.log(outputTwo); let outputOne
= "Try to get this to log only once to the browser console";
console.clear();
console.log(outputOne);
```

3. Use typeof to Check the Type of a Variable

Description

You can use `typeof` to check the data structure, or type, of a variable. This is useful in debugging when working with multiple data types. If you think you're adding two numbers, but one is actually a string, the results can be unexpected. Type errors can lurk in calculations or function calls. Be careful especially when you're accessing and working with external data in the form of a JavaScript Object Notation (JSON) object. Here are some examples using `typeof`:

> **console.log(typeof ""); // outputs "string"**
> **console.log(typeof 0); // outputs "number"**
> **console.log(typeof []); // outputs "object"**
> **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.

Instructions

Add two `console.log()` statements to check the `typeof` each of the two variables `seven` and `three` in the code.

Challenge Seed

```
let seven = 7;
let three = "3";
console.log(seven + three);
// Add your code below this line
```

Solution

```
let seven = 7;let three = "3";console.log(typeof seven);
console.log(typeof three);
```

4. Catch Misspelled Variable and Function Names

Description

The `console.log()` and `typeof` methods are the two primary ways to check intermediate values and types of program output. Now it's time to get into the common forms that bugs take. One syntax-level issue that fast typers can commiserate with is the humble spelling error. Transposed, missing, or mis-capitalized characters in a variable or function name will have the browser looking for an object that doesn't exist - and complain in the form of a reference error. JavaScript variable and function names are case-sensitive.

Instructions

Fix the two spelling errors in the code so the `netWorkingCapital` calculation works.

Challenge Seed

```
let receivables = 10;
let payables = 8;
let netWorkingCapital = recievables - payable;
console.log(`Net working capital is: ${netWorkingCapital}`);
```

Solution

```
// solution required
```

5. Catch Unclosed Parentheses, Brackets, Braces and Quotes

Description

Another syntax error to be aware of is that all opening parentheses, brackets, curly braces, and quotes have a closing pair. Forgetting a piece tends to happen when you're editing existing code and inserting items with one of the pair types. Also, take care when nesting code blocks into others, such as adding a callback function as an argument to a method. One way to avoid this mistake is as soon as the opening character is typed, immediately include the closing match, then move the cursor back between them and continue coding. Fortunately, most modern code editors generate the second half of the pair automatically.

Instructions

Fix the two pair errors in the code.

Challenge Seed

```
let myArray = [1, 2, 3;
let arraySum = myArray.reduce((previous, current =>  previous + current);
console.log(`Sum of array values is: ${arraySum}`);
```

Solution

```
// solution required
```

6. Catch Mixed Usage of Single and Double Quotes

Description

JavaScript allows the use of both single (') and double ("") quotes to declare a string. Deciding which one to use generally comes down to personal preference, with some exceptions. Having two choices is great when a string has contractions or another piece of text that's in quotes. Just be careful that you don't close the string too early, which causes a syntax error. Here are some examples of mixing quotes:

```
// These are correct:
const grouchoContraction = "I've had a perfectly wonderful evening, but this wasn't it.";
const quoteInString = "Groucho Marx once said 'Quote me as saying I was mis-quoted.'";
// This is incorrect:
const uhOhGroucho = 'I've had a perfectly wonderful evening, but this wasn't it.';
```

Of course, it is okay to use only one style of quotes. You can escape the quotes inside the string by using the backslash (\) escape character:

```
// Correct use of same quotes:
const allSameQuotes = 'I\'ve had a perfectly wonderful evening, but this wasn\'t it.';
```

Instructions

Fix the string so it either uses different quotes for the `href` value, or escape the existing ones. Keep the double quote marks around the entire string.

Challenge Seed

```
let innerHtml = "<p>Click here to <a href="#Home">return home</a></p>";
console.log(innerHtml);
```

Solution

```
// solution required
```

7. Catch Use of Assignment Operator Instead of Equality Operator

Description

Branching programs, i.e. ones that do different things if certain conditions are met, rely on `if`, `else if`, and `else` statements in JavaScript. The condition sometimes takes the form of testing whether a result is equal to a value. This logic is spoken (in English, at least) as "if x equals y, then ..." which can literally translate into code using the `=`, or assignment operator. This leads to unexpected control flow in your program. As covered in previous challenges, the assignment operator (`=`) in JavaScript assigns a value to a variable name. And the `==` and `===` operators check for equality (the triple `===` tests for strict equality, meaning both value and type are the same). The code below assigns `x` to be 2, which evaluates as `true`. 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`.

```
let x = 1;
let y = 2;
if (x = y) {
  // this code block will run for any value of y (unless y were originally set as a falsy)
} else {
  // this code block is what should run (but won't) in this example
}
```

Instructions

Fix the condition so the program runs the right branch, and the appropriate value is assigned to `result`.

Challenge Seed

```
let x = 7;
let y = 9;
let result = "to come";

if(x = y) {
  result = "Equal!";
} else {
  result = "Not equal!";
}

console.log(result);
```

Solution

```
// solution required
```

8. Catch Missing Open and Closing Parenthesis After a Function Call

Description

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!"**

Instructions

Fix the code so the variable `result` is set to the value returned from calling the function `getNine`.

Challenge Seed

```
function getNine() {
  let x = 6;
  let y = 3;
  return x + y;
}

let result = getNine;
console.log(result);
```

Solution

```
// solution required
```

9. Catch Arguments Passed in the Wrong Order When Calling a Function

Description

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.

Instructions

The function `raiseToPower` raises a base to an exponent. Unfortunately, it's not called properly - fix the code so the value of `power` is the expected 8.

Challenge Seed

```
function raiseToPower(b, e) {
  return Math.pow(b, e);
}

let base = 2;
let exp = 3;
let power = raiseToPower(exp, base);
console.log(power);
```

Solution

```
// solution required
```

10. Catch Off By One Errors When Using Indexing

Description

`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. JavaScript indexing starts at zero, not one, which means the last index is always one less than the length of the item. If you try to access an index equal to the length, the program may throw an "index out of range" reference error or print `undefined`. When you use string or array methods that take index ranges as arguments, it helps to read the documentation and understand if they are inclusive (the item at the given index is part of what's returned) or not. Here are some examples of off by one errors:

```
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]);
}
```

Instructions

Fix the two indexing errors in the following function so all the numbers 1 through 5 are printed to the console.

Challenge Seed

```
function countToFive() {
  let firstFive = "12345";
  let len = firstFive.length;
  // Fix the line below
  for (let i = 1; i <= len; i++) {
  // Do not alter code below this line
    console.log(firstFive[i]);
  }
}

countToFive();
```

Solution

```
// solution required
```

11. Use Caution When Reinitializing Variables Inside a Loop

Description

Sometimes it's necessary to save information, increment counters, or re-set variables within a loop. A potential issue is when variables either should be reinitialized, and aren't, or vice versa. This is particularly dangerous if you accidentally reset the variable being used for the terminal condition, causing an infinite loop. Printing variable values with each cycle of your loop by using `console.log()` can uncover buggy behavior related to resetting, or failing to reset a variable.

Instructions

The following function is supposed to create a two-dimensional array with `m` rows and `n` columns of zeroes. Unfortunately, it's not producing the expected output because the `row` variable isn't being reinitialized (set back to an empty array) in the outer loop. Fix the code so it returns a correct 3x2 array of zeroes, which looks like `[[0, 0], [0, 0], [0, 0]]`.

Challenge Seed

```
function zeroArray(m, n) {
  // Creates a 2-D array with m rows and n columns of zeroes
  let newArray = [];
  let row = [];
  for (let i = 0; i < m; i++) {
    // Adds the m-th row into newArray

    for (let j = 0; j < n; j++) {
      // Pushes n zeroes into the current row to create the columns
      row.push(0);
    }
    // Pushes the current row, which now has n zeroes in it, to the array
    newArray.push(row);
  }
```

```
  return newArray;
}

let matrix = zeroArray(3, 2);
console.log(matrix);
```

Solution

```
// solution required
```

12. Prevent Infinite Loops with a Valid Terminal Condition

Description

The final topic is the dreaded infinite loop. Loops are great tools when you need your program to run a code block a certain number of times or until a condition is met, but they need a terminal condition that ends the looping. Infinite loops are likely to freeze or crash the browser, and cause general program execution mayhem, which no one wants. There was an example of an infinite loop in the introduction to this section - it had no terminal condition to break out of the `while` loop inside `loopy()`. Do NOT call this function!

```
function loopy() {
  while(true) {
    console.log("Hello, world!");
  }
}
```

It's the programmer's job to ensure that the terminal condition, which tells the program when to break out of the loop code, is eventually reached. One error is incrementing or decrementing a counter variable in the wrong direction from the terminal condition. Another one is accidentally resetting a counter or index variable within the loop code, instead of incrementing or decrementing it.

Instructions

The `myFunc()` function contains an infinite loop because the terminal condition `i != 4` will never evaluate to `false` (and break the looping) - `i` will increment by 2 each pass, and jump right over 4 since `i` is odd to start. Fix the comparison operator in the terminal condition so the loop only runs for `i` less than or equal to 4.

Challenge Seed

```
function myFunc() {
  for (let i = 1; i != 4; i += 2) {
    console.log("Still going!");
  }
}
```

Solution

```
// solution required
```

Basic Data Structures

1. Use an Array to Store a Collection of Data

Description

The below is an example of the simplest implementation of an array data structure. This is known as a one-dimensional array, meaning it only has one level, or that it does not have any other arrays nested within it. Notice it contains booleans, strings, and numbers, among other valid JavaScript data types:

```
let simpleArray = ['one', 2, 'three', true, false, undefined, null];
console.log(simpleArray.length);
// logs 7
```

All arrays have a length property, which as shown above, can be very easily accessed with the syntax `Array.length`. A more complex implementation of an array can be seen below. This is known as a multi-dimensional array, or an array that contains other arrays. Notice that this array also contains JavaScript objects, which we will examine very closely in our next section, but for now, all you need to know is that arrays are also capable of storing complex objects.

```
let complexArray = [
  [
    {
      one: 1,
      two: 2
    },
    {
      three: 3,
      four: 4
    }
  ],
  [
    {
      a: "a",
      b: "b"
    },
    {
      c: "c",
      d: "d"
    }
  ]
];
```

Instructions

We have defined a variable called `yourArray`. Complete the statement by assigning an array of at least 5 elements in length to the `yourArray` variable. Your array should contain at least one string, one number, and one boolean.

Challenge Seed

```
let yourArray; // change this line
```

Solution

```
// solution required
```

2. Access an Array's Contents Using Bracket Notation

Description

The fundamental feature of any data structure is, of course, the ability to not only store data, but to be able to retrieve that data on command. So, now that we've learned how to create an array, let's begin to think about how we can access that array's information. When we define a simple array as seen below, there are 3 items in it:

> **let ourArray = ["a", "b", "c"];**

In an array, each array item has an index. This index doubles as the position of that item in the array, and how you reference it. However, it is important to note, that JavaScript arrays are zero-indexed, meaning that the first element of an array is actually at the *zeroth* position, not the first. In order to retrieve an element from an array we can enclose an index in brackets and append it to the end of an array, or more commonly, to a variable which references an array object. This is known as bracket notation. For example, if we want to retrieve the `"a"` from `ourArray` and assign it to a variable, we can do so with the following code:

> **let ourVariable = ourArray[0];**
> **// ourVariable equals "a"**

In addition to accessing the value associated with an index, you can also *set* an index to a value using the same notation:

> **ourArray[1] = "not b anymore";**
> **// ourArray now equals ["a", "not b anymore", "c"];**

Using bracket notation, we have now reset the item at index 1 from `"b"` , to `"not b anymore"` .

Instructions

In order to complete this challenge, set the 2nd position (index `1`) of `myArray` to anything you want, besides `"b"` .

Challenge Seed

```
let myArray = ["a", "b", "c", "d"];
// change code below this line

//change code above this line
console.log(myArray);
```

Solution

```
// solution required
```

3. Add Items to an Array with push() and unshift()

Description

An array's length, like the data types it can contain, is not fixed. Arrays can be defined with a length of any number of elements, and elements can be added or removed over time; in other words, arrays are mutable. In this challenge, we will look at two methods with which we can programmatically modify an array: `Array.push()` and `Array.unshift()` . Both methods take one or more elements as parameters and add those elements to the array the method is being called on; the `push()` method adds elements to the end of an array, and `unshift()` adds elements to the beginning. Consider the following:

> **let twentyThree = 'XXIII';**
> **let romanNumerals = ['XXI', 'XXII'];**
>
> **romanNumerals.unshift('XIX', 'XX');**
> **// now equals ['XIX', 'XX', 'XXI', 'XXII']**
>
> **romanNumerals.push(twentyThree);**
> **// now equals ['XIX', 'XX', 'XXI', 'XXII', 'XXIII'] Notice that we can also pass variables, which allows us even greater flexibility in dynamically modifying our array's data.**

Instructions

We have defined a function, `mixedNumbers`, which we are passing an array as an argument. Modify the function by using `push()` and `unshift()` to add `'I', 2, 'three'` to the beginning of the array and `7, 'VIII', 9` to the end so that the returned array contains representations of the numbers 1-9 in order.

Challenge Seed

```
function mixedNumbers(arr) {
  // change code below this line

  // change code above this line
  return arr;
}

// do not change code below this line
console.log(mixedNumbers(['IV', 5, 'six']));
```

Solution

```
// solution required
```

4. Remove Items from an Array with pop() and shift()

Description

Both `push()` and `unshift()` have corresponding methods that are nearly functional opposites: `pop()` and `shift()`. As you may have guessed by now, instead of adding, `pop()` *removes* an element from the end of an array, while `shift()` removes an element from the beginning. 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 []**

Instructions

We have defined a function, `popShift`, which takes an array as an argument and returns a new array. Modify the function, using `pop()` and `shift()`, to remove the first and last elements of the argument array, and assign the removed elements to their corresponding variables, so that the returned array contains their values.

Challenge Seed

```
function popShift(arr) {
  let popped; // change this line
  let shifted; // change this line
  return [shifted, popped];
}
```

```
// do not change code below this line
console.log(popShift(['challenge', 'is', 'not', 'complete']));
```

Solution

```
// solution required
```

5. Remove Items Using splice()

Description

Ok, so we've learned how to remove elements from the beginning and end of arrays using `shift()` and `pop()`, but what if we want to remove an element from somewhere in the middle? Or remove more than one element at once? Well, that's where `splice()` comes in. `splice()` allows us to do just that: remove any number of consecutive elements from anywhere in an array. `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. And remember, arrays are *zero-indexed*, so to indicate the first element of an array, we would use `0`. `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']
```

`splice()` not only modifies the array it's being called on, but it also returns a new array containing the value of the removed elements:

```
let array = ['I', 'am', 'feeling', 'really', 'happy'];

let newArray = array.splice(3, 2);
// newArray equals ['really', 'happy']
```

Instructions

We've defined a function, `sumOfTen`, which takes an array as an argument and returns the sum of that array's elements. Modify the function, using `splice()`, so that it returns a value of `10`.

Challenge Seed

```
function sumOfTen(arr) {
  // change code below this line

  // change code above this line
  return arr.reduce((a, b) => a + b);
}

// do not change code below this line
console.log(sumOfTen([2, 5, 1, 5, 2, 1]));
```

Solution

```
// solution required
```

6. Add Items Using splice()

Description

Remember in the last challenge we mentioned that `splice()` can take up to three parameters? Well, we can go one step further with `splice()` — 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. For instance, let's say you're storing a color scheme for a set of DOM elements in an array, and want to dynamically change a color based on some action:

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

let colorScheme = ['#878787', '#a08794', '#bb7e8c', '#c9b6be', '#d1becf'];

colorScheme = colorChange(colorScheme, 2, '#332327');
// we have removed '#bb7e8c' and added '#332327' in its place
// colorScheme now equals ['#878787', '#a08794', '#332327', '#c9b6be', '#d1becf']
```

This function takes an array of hex values, an index at which to remove an element, and the new color to replace the removed element with. The return value is an array containing a newly modified color scheme! While this example is a bit oversimplified, we can see the value that utilizing `splice()` to its maximum potential can have.

Instructions

We have defined a function, `htmlColorNames`, which takes an array of HTML colors as an argument. Modify the function using `splice()` to remove the first two elements of the array and add `'DarkSalmon'` and `'BlanchedAlmond'` in their respective places.

Challenge Seed

```
function htmlColorNames(arr) {
  // change code below this line

  // change code above this line
  return arr;
}

// do not change code below this line
console.log(htmlColorNames(['DarkGoldenRod', 'WhiteSmoke', 'LavenderBlush', 'PaleTurqoise',
'FireBrick']));
```

Solution

```
// solution required
```

7. Copy Array Items Using slice()

Description

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.

Instructions

We have defined a function, `forecast`, that takes an array as an argument. Modify the function using `slice()` to extract information from the argument array and return a new array that contains the elements `'warm'` and `'sunny'`.

Challenge Seed

```
function forecast(arr) {
  // change code below this line

  return arr;
}

// do not change code below this line
console.log(forecast(['cold', 'rainy', 'warm', 'sunny', 'cool', 'thunderstorms']));
```

Solution

```
// solution required
```

8. Copy an Array with the Spread Operator

Description

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: `...` In practice, we can use the spread operator to copy an array like so:

> let thisArray = [true, true, undefined, false, null];
> let thatArray = [...thisArray];
> // thatArray equals [true, true, undefined, false, null]
> // thisArray remains unchanged, and is identical to thatArray

Instructions

We have defined a function, `copyMachine` which takes `arr` (an array) and `num` (a number) as arguments. The function is supposed to return a new array made up of `num` copies of `arr`. We have done most of the work for you, but it doesn't work quite right yet. Modify the function using spread syntax so that it works correctly (hint: another method we have already covered might come in handy here!).

Challenge Seed

```
function copyMachine(arr, num) {
  let newArr = [];
  while (num >= 1) {
    // change code below this line

    // change code above this line
    num--;
  }
  return newArr;
}
```

```
// change code here to test different cases:
console.log(copyMachine([true, false, true], 2));
```

Solution

```
// solution required
```

9. Combine Arrays with the Spread Operator

Description

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 complex and more verbose had we used traditional methods.

Instructions

We have defined a function `spreadOut` that returns the variable `sentence`. Modify the function using the spread operator so that it returns the array `['learning', 'to', 'code', 'is', 'fun']`.

Challenge Seed

```
function spreadOut() {
  let fragment = ['to', 'code'];
  let sentence; // change this line
  return sentence;
}

// do not change code below this line
console.log(spreadOut());
```

Solution

```
// solution required
```

10. Check For The Presence of an Element With indexOf()

Description

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 `-1` if 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
```

Instructions

`indexOf()` can be incredibly useful for quickly checking for the presence of an element on an array. We have defined a function, `quickCheck`, that takes an array and an element as arguments. Modify the function using `indexOf()` so that it returns `true` if the passed element exists on the array, and `false` if it does not.

Challenge Seed

```
function quickCheck(arr, elem) {
  // change code below this line

  // change code above this line
}

// change code here to test different cases:
console.log(quickCheck(['squash', 'onions', 'shallots'], 'mushrooms'));
```

Solution

```
// solution required
```

11. Iterate Through All an Array's Items Using For Loops

Description

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 `for` loop. 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.

Instructions

We have defined a function, `filteredArray`, which takes `arr`, a nested array, and `elem` as arguments, and returns a new array. `elem` represents an element that may or may not be present on one or more of the arrays

nested within `arr`. Modify the function, using a `for` loop, to return a filtered version of the passed array such that any array nested within `arr` containing `elem` has been removed.

Challenge Seed

```
function filteredArray(arr, elem) {
  let newArr = [];
  // change code below this line

  // change code above this line
  return newArr;
}

// change code here to test different cases:
console.log(filteredArray([[3, 2, 3], [1, 6, 3], [3, 13, 26], [19, 3, 9]], 3));
```

Solution

```
// solution required
```

12. Create complex multi-dimensional arrays

Description

Awesome! You have just learned a ton about arrays! This has been a fairly high level overview, and there is plenty more to learn about working with arrays, much of which you will see in later sections. But before moving on to looking at Objects, lets take one more look, and see how arrays can become a bit more complex than what we have seen in previous challenges. One of the most powerful features when thinking of arrays as data structures, is that arrays can contain, or even be completely made up of other arrays. We have seen arrays that contain arrays in previous challenges, but fairly simple ones. However, arrays can contain an infinite depth of arrays that can contain other arrays, each with their own arbitrary levels of depth, and so on. In this way, an array can very quickly become very complex data structure, known as a multi-dimensional, or nested array. Consider the following example:

```
let nestedArray = [ // top, or first level - the outer most array
  ['deep'], // an array within an array, 2 levels of depth
  [
    ['deeper'], ['deeper'] // 2 arrays nested 3 levels deep
  ],
  [
    [
      ['deepest'], ['deepest'] // 2 arrays nested 4 levels deep
    ],
    [
      [
        ['deepest-est?'] // an array nested 5 levels deep
      ]
    ]
  ]
];
```

While this example may seem convoluted, this level of complexity is not unheard of, or even unusual, when dealing with large amounts of data. However, we can still very easily access the deepest levels of an array this complex with bracket notation:

```
console.log(nestedArray[2][1][0][0][0]);
// logs: deepest-est?
```

And now that we know where that piece of data is, we can reset it if we need to:

```
nestedArray[2][1][0][0][0] = 'deeper still';

console.log(nestedArray[2][1][0][0][0]);
// now logs: deeper still
```

Instructions

We have defined a variable, `myNestedArray`, set equal to an array. Modify `myNestedArray`, using any combination of strings, numbers, and booleans for data elements, so that it has exactly five levels of depth (remember, the outer-most array is level 1). Somewhere on the third level, include the string `'deep'`, on the fourth level, include the string `'deeper'`, and on the fifth level, include the string `'deepest'`.

Challenge Seed

```
let myNestedArray = [
  // change code below this line
  ['unshift', false, 1, 2, 3, 'complex', 'nested'],
  ['loop', 'shift', 6, 7, 1000, 'method'],
  ['concat', false, true, 'spread', 'array'],
  ['mutate', 1327.98, 'splice', 'slice', 'push'],
  ['iterate', 1.3849, 7, '8.4876', 'arbitrary', 'depth']
  // change code above this line
];
```

Solution

```
// solution required
```

13. Add Key-Value Pairs to JavaScript Objects

Description

At their most basic, objects are just collections of key-value pairs, or in other words, pieces of data mapped to unique identifiers that we call properties or keys. Let's take a look at a very simple example:

```
let FCC_User = {
  username: 'awesome_coder',
  followers: 572,
  points: 1741,
  completedProjects: 15
};
```

The above code defines an object called `FCC_User` that has four properties, each of which map to a specific value. If we wanted to know the number of `followers` `FCC_User` has, we can access that property by writing:

```
let userData = FCC_User.followers;
// userData equals 572
```

This is called dot notation. Alternatively, we can also access the property with brackets, like so:

```
let userData = FCC_User['followers']
// userData equals 572
```

Notice that with bracket notation, we enclosed `followers` in quotes. This is because the brackets actually allow us to pass a variable in to be evaluated as a property name (hint: keep this in mind for later!). Had we passed `followers` in without the quotes, the JavaScript engine would have attempted to evaluate it as a variable, and a `ReferenceError: followers is not defined` would have been thrown.

Instructions

Using the same syntax, we can also *add new* key-value pairs to objects. We've created a `foods` object with three entries. Add three more entries: `bananas` with a value of `13`, `grapes` with a value of `35`, and `strawberries` with a value of `27`.

Challenge Seed

```
let foods = {
  apples: 25,
  oranges: 32,
  plums: 28
};

// change code below this line

// change code above this line

console.log(foods);
```

Solution

```
// solution required
```

14. Modify an Object Nested Within an Object

Description

Now let's take a look at a slightly more complex object. Object properties can be nested to an arbitrary depth, and their values can be any type of data supported by JavaScript, including arrays and even other objects. Consider the following:

```
let nestedObject = {
  id: 28802695164,
  date: 'December 31, 2016',
  data: {
    totalUsers: 99,
    online: 80,
    onlineStatus: {
      active: 67,
      away: 13
    }
  }
};
```

`nestedObject` has three unique keys: `id`, whose value is a number, `date` whose value is a string, and `data`, whose value is an object which has yet another object nested within it. While structures can quickly become complex, we can still use the same notations to access the information we need.

Instructions

Here we've defined an object, `userActivity`, which includes another object nested within it. You can modify properties on this nested object in the same way you modified properties in the last challenge. Set the value of the `online` key to `45`.

Challenge Seed

```
let userActivity = {
  id: 23894201352,
  date: 'January 1, 2017',
  data: {
```

```
    totalUsers: 51,
    online: 42
  }
};

// change code below this line

// change code above this line

console.log(userActivity);
```

Solution

```
// solution required
```

15. Access Property Names with Bracket Notation

Description

In the first object challenge we mentioned the use of bracket notation as a way to access property values using the evaluation of a variable. For instance, imagine that our `foods` object is being used in a program for a supermarket cash register. We have some function that sets the `selectedFood` and we want to check our `foods` object for the presence of that food. This might look like:

> **let selectedFood = getCurrentFood(scannedItem);**
> **let inventory = foods[selectedFood];**

This code will evaluate the value stored in the `selectedFood` variable and return the value of that key in the `foods` object, or `undefined` if it is not present. Bracket notation is very useful because sometimes object properties are not known before runtime or we need to access them in a more dynamic way.

Instructions

We've defined a function, `checkInventory`, which receives a scanned item as an argument. Return the current value of the `scannedItem` key in the `foods` object. You can assume that only valid keys will be provided as an argument to `checkInventory`.

Challenge Seed

```
let foods = {
  apples: 25,
  oranges: 32,
  plums: 28,
  bananas: 13,
  grapes: 35,
  strawberries: 27
};
// do not change code above this line

function checkInventory(scannedItem) {
  // change code below this line

}

// change code below this line to test different cases:
console.log(checkInventory("apples"));
```

Solution

```
// solution required
```

16. Use the delete Keyword to Remove Object Properties

Description

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. In earlier challenges, we have both added to and modified an object's key-value pairs. Here we will see how we can *remove* a key-value pair from an object. Let's revisit our `foods` object example one last time. If we wanted to remove the `apples` key, we can remove it by using the `delete` keyword like this:

> **delete foods.apples;**

Instructions

Use the delete keyword to remove the `oranges`, `plums`, and `strawberries` keys from the `foods` object.

Challenge Seed

```
let foods = {
  apples: 25,
  oranges: 32,
  plums: 28,
  bananas: 13,
  grapes: 35,
  strawberries: 27
};

// change code below this line

// change code above this line

console.log(foods);
```

Solution

```
// solution required
let foods = {
  apples: 25,
  oranges: 32,
  plums: 28,
  bananas: 13,
  grapes: 35,
  strawberries: 27
};

delete foods.oranges;
delete foods.plums;
delete foods.strawberries;

console.log(foods);
```

17. Check if an Object has a Property

Description

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 `users` with 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**

Instructions

We've created an object, `users`, with some users in it and a function `isEveryoneHere`, which we pass the `users` object to as an argument. Finish writing this function so that it returns `true` only if the `users` object contains all four names, `Alan`, `Jeff`, `Sarah`, and `Ryan`, as keys, and `false` otherwise.

Challenge Seed

```
let users = {
  Alan: {
    age: 27,
    online: true
  },
  Jeff: {
    age: 32,
    online: true
  },
  Sarah: {
    age: 48,
    online: true
  },
  Ryan: {
    age: 19,
    online: true
  }
};

function isEveryoneHere(obj) {
  // change code below this line

  // change code above this line
}

console.log(isEveryoneHere(users));
```

Solution

```
let users = {
  Alan: {
    age: 27,
    online: true
  },
  Jeff: {
    age: 32,
    online: true
  },
  Sarah: {
    age: 48,
    online: true
  },
  Ryan: {
    age: 19,
    online: true
  }
};

function isEveryoneHere(obj) {
  return [
    'Alan',
    'Jeff',
    'Sarah',
    'Ryan'
  ].every(i => obj.hasOwnProperty(i));
```

```
}

console.log(isEveryoneHere(users));
```

18. Iterate Through the Keys of an Object with a for...in Statement

Description

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 `users` object, 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 key's position on an object, or the relative order in which it appears, is irrelevant when referencing or accessing that key.

Instructions

We've defined a function, `countOnline` ; use a for...in statement within this function to loop through the users in the `users` object and return the number of users whose `online` property is set to `true` .

Challenge Seed

```
let users = {
  Alan: {
    age: 27,
    online: false
  },
  Jeff: {
    age: 32,
    online: true
  },
  Sarah: {
    age: 48,
    online: false
  },
  Ryan: {
    age: 19,
    online: true
  }
};

function countOnline(obj) {
  // change code below this line

  // change code above this line
}

console.log(countOnline(users));
```

Solution

```
let users = {
  Alan: {
    age: 27,
    online: false
  },
  Jeff: {
    age: 32,
    online: true
  },
  Sarah: {
    age: 48,
    online: false
  },
  Ryan: {
    age: 19,
    online: true
  }
};

function countOnline(obj) {
  let online = 0;
  for(let user in obj){
    if(obj[user].online == true) {
      online += 1;
    }
  }
  return online;
}

console.log(countOnline(users));
```

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

Description

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.

Instructions

Finish writing the `getArrayOfUsers` function so that it returns an array containing all the properties in the object it receives as an argument.

Challenge Seed

```
let users = {
  Alan: {
    age: 27,
    online: false
  },
  Jeff: {
    age: 32,
    online: true
  },
  Sarah: {
    age: 48,
    online: false
  },
  Ryan: {
    age: 19,
    online: true
```

```
  }
};

function getArrayOfUsers(obj) {
  // change code below this line

  // change code above this line
}

console.log(getArrayOfUsers(users));
```

Solution

```
let users = {
  Alan: {
    age: 27,
    online: false
  },
  Jeff: {
    age: 32,
    online: true
  },
  Sarah: {
    age: 48,
    online: false
  },
  Ryan: {
    age: 19,
    online: true
  }
};

function getArrayOfUsers(obj) {
  return Object.keys(users);
}

console.log(getArrayOfUsers(users));
```

20. Modify an Array Stored in an Object

Description

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!

Instructions

Take a look at the object we've provided in the code editor. The `user` object contains three keys. The `data` key contains five keys, one of which contains an array of `friends`. From this, you can see how flexible objects are as data structures. We've started writing a function `addFriend`. Finish writing it so that it takes a `user` object and adds the name of the `friend` argument to the array stored in `user.data.friends` and returns that array.

Challenge Seed

```
let user = {
  name: 'Kenneth',
  age: 28,
  data: {
    username: 'kennethCodesAllDay',
```

```
    joinDate: 'March 26, 2016',
    organization: 'freeCodeCamp',
    friends: [
      'Sam',
      'Kira',
      'Tomo'
    ],
    location: {
      city: 'San Francisco',
      state: 'CA',
      country: 'USA'
    }
  }
};

function addFriend(userObj, friend) {
  // change code below this line

  // change code above this line
}

console.log(addFriend(user, 'Pete'));
```

Solution

```
// solution required
```

Basic Algorithm Scripting

1. Convert Celsius to Fahrenheit

Description

The algorithm to convert from Celsius to Fahrenheit is the temperature in Celsius times `9/5`, plus `32`. You are given a variable `celsius` representing a temperature in Celsius. Use the variable `fahrenheit` already defined and assign it the Fahrenheit temperature equivalent to the given Celsius temperature. Use the algorithm mentioned above to help convert the Celsius temperature to Fahrenheit. Don't worry too much about the function and return statements as they will be covered in future challenges. For now, only use operators that you have already learned.

Instructions

Challenge Seed

```
function convertToF(celsius) {
  let fahrenheit;
  return fahrenheit;
}

convertToF(30);
```

Solution

```
function convertToF(celsius) {
  let fahrenheit = celsius * 9/5 + 32;

  return fahrenheit;
}
```

```
convertToF(30);
```

2. Reverse a String

Description

Reverse the provided string. You may need to turn the string into an array before you can reverse it. Your result must be a string. Remember to use Read-Search-Ask if you get stuck. Write your own code.

Instructions

Challenge Seed

```
function reverseString(str) {
  return str;
}

reverseString("hello");
```

Solution

```
function reverseString(str) {
  return str.split('').reverse().join('');
}

reverseString("hello");
```

3. Factorialize a Number

Description

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` Only integers greater than or equal to zero will be supplied to the function. Remember to use Read-Search-Ask if you get stuck. Write your own code.

Instructions

Challenge Seed

```
function factorialize(num) {
  return num;
}

factorialize(5);
```

Solution

```
function factorialize(num) {
  return num < 1 ? 1 : num * factorialize(num - 1);
}
```

```
factorialize(5);
```

4. Find the Longest Word in a String

Description

Return the length of the longest word in the provided sentence. Your response should be a number. Remember to use Read-Search-Ask if you get stuck. Write your own code.

Instructions

Challenge Seed

```
function findLongestWordLength(str) {
  return str.length;
}

findLongestWordLength("The quick brown fox jumped over the lazy dog");
```

Solution

```
function findLongestWordLength(str) {
  return str.split(' ').sort((a, b) => b.length - a.length)[0].length;
}

findLongestWordLength("The quick brown fox jumped over the lazy dog");
```

5. Return Largest Numbers in Arrays

Description

Return an array consisting of the largest number from each provided sub-array. For simplicity, the provided array will contain exactly 4 sub-arrays. Remember, you can iterate through an array with a simple for loop, and access each member with array syntax `arr[i]` . Remember to use Read-Search-Ask if you get stuck. Write your own code.

Instructions

Challenge Seed

```
function largestOfFour(arr) {
  // You can do this!
  return arr;
}

largestOfFour([[4, 5, 1, 3], [13, 27, 18, 26], [32, 35, 37, 39], [1000, 1001, 857, 1]]);
```

Solution

```
function largestOfFour(arr) {
  return arr.map(subArr => Math.max.apply(null, subArr));
}
```

```
largestOfFour([[4, 5, 1, 3], [13, 27, 18, 26], [32, 35, 37, 39], [1000, 1001, 857, 1]]);
```

6. Confirm the Ending

Description

Check if a string (first argument, `str`) ends with the given target string (second argument, `target`). This challenge *can* be solved with the `.endsWith()` method, which was introduced in ES2015. But for the purpose of this challenge, we would like you to use one of the JavaScript substring methods instead. Remember to use Read-Search-Ask if you get stuck. Write your own code.

Instructions

Challenge Seed

```
function confirmEnding(str, target) {
  // "Never give up and good luck will find you."
  // -- Falcor
  return str;
}

confirmEnding("Bastian", "n");
```

Solution

```
function confirmEnding(str, target) {
  return str.substring(str.length - target.length) === target;
}

confirmEnding("Bastian", "n");
```

7. Repeat a String Repeat a String

Description

Repeat a given string `str` (first argument) for `num` times (second argument). Return an empty string if `num` is not a positive number. Remember to use Read-Search-Ask if you get stuck. Write your own code.

Instructions

Challenge Seed

```
function repeatStringNumTimes(str, num) {
  // repeat after me
  return str;
}

repeatStringNumTimes("abc", 3);
```

Solution

```
function repeatStringNumTimes(str, num) {
  if (num < 0) return '';
  return num === 1 ? str : str + repeatStringNumTimes(str, num-1);
}

repeatStringNumTimes("abc", 3);
```

8. Truncate a String

Description

Truncate a string (first argument) if it is longer than the given maximum string length (second argument). Return the truncated string with a `...` ending. Remember to use Read-Search-Ask if you get stuck. Write your own code.

Instructions

Challenge Seed

```
function truncateString(str, num) {
  // Clear out that junk in your trunk
  return str;
}

truncateString("A-tisket a-tasket A green and yellow basket", 8);
```

Solution

```
function truncateString(str, num) {
  if (num >= str.length) {
    return str;
  }

  return str.slice(0, num) + '...';
}

truncateString("A-tisket a-tasket A green and yellow basket", 8);
```

9. Finders Keepers

Description

Create a function that looks through an array (first argument) and returns the first element in the array that passes a truth test (second argument). If no element passes the test, return undefined. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function findElement(arr, func) {
  let num = 0;
  return num;
}

findElement([1, 2, 3, 4], num => num % 2 === 0);
```

Solution

```
function findElement(arr, func) {
  return arr.filter(func)[0];
}

findElement([1, 2, 3, 4], num => num % 2 === 0);
```

10. Boo who

Description

Check if a value is classified as a boolean primitive. Return true or false. Boolean primitives are true and false. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function booWho(bool) {
  // What is the new fad diet for ghost developers? The Boolean.
  return bool;
}

booWho(null);
```

Solution

```
function booWho(bool) {
  return typeof bool === "boolean";
}

booWho(null);
```

11. Title Case a Sentence

Description

Return the provided string with the first letter of each word capitalized. Make sure the rest of the word is in lower case. For the purpose of this exercise, you should also capitalize connecting words like "the" and "of". Remember to use Read-Search-Ask if you get stuck. Write your own code.

Instructions

Challenge Seed

```
function titleCase(str) {
  return str;
}

titleCase("I'm a little tea pot");
```

Solution

```
function titleCase(str) {
  return str.split(' ').map(word => word.charAt(0).toUpperCase() +
word.substring(1).toLowerCase()).join(' ');
}

titleCase("I'm a little tea pot");
```

12. Slice and Splice

Description

You are given two arrays and an index. Use the array methods `slice` and `splice` to copy each element of the first array into the second array, in order. Begin inserting elements at index `n` of the second array. Return the resulting array. The input arrays should remain the same after the function runs. Remember to use Read-Search-Ask if you get stuck. Write your own code.

Instructions

Challenge Seed

```
function frankenSplice(arr1, arr2, n) {
  // It's alive. It's alive!
  return arr2;
}

frankenSplice([1, 2, 3], [4, 5, 6], 1);
```

After Test

```
let testArr1 = [1, 2];
let testArr2 = ["a", "b"];
```

Solution

```
function frankenSplice(arr1, arr2, n) {
  // It's alive. It's alive!
  let result = arr2.slice();
  for (let i = 0; i < arr1.length; i++) {
    result.splice(n+i, 0, arr1[i]);
  }
  return result;
}

frankenSplice([1, 2, 3], [4, 5], 1);
```

13. Falsy Bouncer

Description

Remove all falsy values from an array. Falsy values in JavaScript are `false`, `null`, `0`, `""`, `undefined`, and `NaN`. Hint: Try converting each value to a Boolean. Remember to use Read-Search-Ask if you get stuck. Write your own code.

Instructions

Challenge Seed

```
function bouncer(arr) {
  // Don't show a false ID to this bouncer.
  return arr;
}

bouncer([7, "ate", "", false, 9]);
```

Solution

```
function bouncer(arr) {
  return arr.filter(e => e);
}

bouncer([7, "ate", "", false, 9]);
```

14. Where do I Belong

Description

Return the lowest index at which a value (second argument) should be inserted into an array (first argument) once it has been sorted. The returned value should be a number. For example, `getIndexToIns([1,2,3,4], 1.5)` **should return** `1` **because it is greater than** `1` **(index 0), but less than** `2` **(index 1). Likewise,** `getIndexToIns([20,3,5], 19)` **should return** `2` **because once the array has been sorted it will look like** `[3,5,20]` **and** `19` **is less than** `20` **(index 2) and greater than** `5` **(index 1). Remember to use Read-Search-Ask if you get stuck. Write your own code.**

Instructions

Challenge Seed

```
function getIndexToIns(arr, num) {
  // Find my place in this sorted array.
  return num;
}

getIndexToIns([40, 60], 50);
```

Solution

```
function getIndexToIns(arr, num) {
  arr = arr.sort((a, b) => a - b);

  for (let i = 0; i < arr.length; i++) {
    if (arr[i] >= num) {
      return i;
    }
  }

  return arr.length;
}

getIndexToIns([40, 60], 50);
```

15. Mutations

Description

Return true if the string in the first element of the array contains all of the letters of the string in the second element of the array. For example, `["hello", "Hello"]`, should return true because all of the letters in the second string are present in the first, ignoring case. The arguments `["hello", "hey"]` should return false because the string "hello" does not contain a "y". Lastly, `["Alien", "line"]`, should return true because all of the letters in "line" are present in "Alien". Remember to use Read-Search-Ask if you get stuck. Write your own code.

Instructions

Challenge Seed

```
function mutation(arr) {
  return arr;
}

mutation(["hello", "hey"]);
```

Solution

```
function mutation(arr) {
  let hash = Object.create(null);

  arr[0].toLowerCase().split('').forEach(c => hash[c] = true);

  return !arr[1].toLowerCase().split('').filter(c => !hash[c]).length;
}

mutation(["hello", "hey"]);
```

16. Chunky Monkey

Description

Write a function that splits an array (first argument) into groups the length of `size` (second argument) and returns them as a two-dimensional array. Remember to use Read-Search-Ask if you get stuck. Write your own code.

Instructions

Challenge Seed

```
function chunkArrayInGroups(arr, size) {
  // Break it up.
  return arr;
}

chunkArrayInGroups(["a", "b", "c", "d"], 2);
```

Solution

```
function chunkArrayInGroups(arr, size) {
  let out = [];
```

```
  for (let i = 0; i < arr.length; i += size) {
    out.push(arr.slice(i, i + size));
  }

  return out;
}

chunkArrayInGroups(["a", "b", "c", "d"], 2);
```

Object Oriented Programming

1. Create a Basic JavaScript Object

Description

Think about things people see everyday, like cars, shops, and birds. These are all `objects`: tangible things people can observe and interact with. What are some qualities of these `objects`? A car has wheels. Shops sell items. Birds have wings. These qualities, or `properties`, define what makes up an `object`. Note that similar `objects` share the same `properties`, but may have different values for those `properties`. For example, all cars have wheels, but not all cars have the same number of wheels. `Objects` in JavaScript are used to model real-world objects, giving them `properties` and behavior just like their real-world counterparts. Here's an example using these concepts to create a `duck` `object`:

```
let duck = {
  name: "Aflac",
  numLegs: 2
};
```

This `duck` `object` has two property/value pairs: a `name` of "Aflac" and a `numLegs` of 2.

Instructions

Create a `dog` `object` with `name` and `numLegs` properties, and set them to a string and a number, respectively.

Challenge Seed

```
let dog = {

};
```

Solution

```
let dog = {
  name: '',
  numLegs: 4
};
```

2. Use Dot Notation to Access the Properties of an Object

Description

The last challenge created an `object` with various `properties`, now you'll see how to access the values of those `properties`. Here's an example:

```
let duck = {
  name: "Aflac",
  numLegs: 2
};
console.log(duck.name);
// This prints "Aflac" to the console
```

Dot notation is used on the `object` name, `duck`, followed by the name of the `property`, `name`, to access the value of "Aflac".

Instructions

Print both `properties` of the `dog` object below to your console.

Challenge Seed

```
let dog = {
  name: "Spot",
  numLegs: 4
};
// Add your code below this line
```

Solution

```
let dog = {
  name: "Spot",
  numLegs: 4
};
console.log(dog.name);
console.log(dog.numLegs);
```

3. Create a Method on an Object

Description

`Objects` can have a special type of `property`, called a `method`. `Methods` are `properties` that are functions. This adds different behavior to an `object`. Here is the `duck` example with a method:

```
let duck = {
  name: "Aflac",
  numLegs: 2,
  sayName: function() {return "The name of this duck is " + duck.name + ".";}
};
duck.sayName();
// Returns "The name of this duck is Aflac."
```

The example adds the `sayName method`, which is a function that returns a sentence giving the name of the `duck`. Notice that the `method` accessed the `name` property in the return statement using `duck.name`. The next challenge will cover another way to do this.

Instructions

Using the `dog object`, give it a method called `sayLegs`. The method should return the sentence "This dog has 4 legs."

Challenge Seed

```
let dog = {
  name: "Spot",
  numLegs: 4,

};

dog.sayLegs();
```

Solution

```
let dog = {
  name: "Spot",
  numLegs: 4,
  sayLegs () {
    return 'This dog has ' + this.numLegs + ' legs.';
  }
};

dog.sayLegs();
```

4. Make Code More Reusable with the this Keyword

Description

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:

> **let duck = {**
> **name: "Aflac",**
> **numLegs: 2,**
> **sayName: function() {return "The name of this duck is " + this.name + ".";}**
> **};**

`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.

Instructions

Modify the `dog.sayLegs` method to remove any references to `dog`. Use the `duck` example for guidance.

Challenge Seed

```
let dog = {
  name: "Spot",
  numLegs: 4,
  sayLegs: function() {return "This dog has " + dog.numLegs + " legs.";}
};

dog.sayLegs();
```

Solution

```
let dog = {
  name: "Spot",
  numLegs: 4,
```

```
  sayLegs () {
    return 'This dog has ' + this.numLegs + ' legs.';
  }
};

dog.sayLegs();
```

5. Define a Constructor Function

Description

`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 `numLegs` set 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.

Instructions

Create a `constructor`, `Dog`, with properties `name`, `color`, and `numLegs` that are set to a string, a string, and a number, respectively.

Challenge Seed

Solution

```
function Dog (name, color, numLegs) {
  this.name = 'name';
  this.color = 'color';
  this.numLegs = 4;
}
```

6. Use a Constructor to Create Objects

Description

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, `this` inside the constructor would not point to the newly created object, giving unexpected results. Now `blueBird` has all the properties defined inside the `Bird` constructor:

```
blueBird.name; // => Albert
blueBird.color; // => blue
blueBird.numLegs; // => 2
```

Just like any other object, its properties can be accessed and modified:

```
blueBird.name = 'Elvira';
blueBird.name; // => Elvira
```

Instructions

Use the `Dog` constructor from the last lesson to create a new instance of `Dog`, assigning it to a variable `hound`.

Challenge Seed

```
function Dog() {
  this.name = "Rupert";
  this.color = "brown";
  this.numLegs = 4;
}
// Add your code below this line
```

Solution

```
function Dog() {
  this.name = "Rupert";
  this.color = "brown";
  this.numLegs = 4;
}
const hound = new Dog();
```

7. Extend Constructors to Receive Arguments

Description

The `Bird` and `Dog` constructors from last challenge worked well. However, notice that all `Birds` that are created with the `Bird` constructor are automatically named Albert, are blue in color, and have two legs. What if you want birds with different values for name and color? It's possible to change the properties of each bird manually but that would be a lot of work:

```
let swan = new Bird();
swan.name = "Carlos";
swan.color = "white";
```

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 `Bird` constructor: `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.

Instructions

Create another `Dog` constructor. This time, set it up to take the parameters `name` and `color`, and have the property `numLegs` fixed at 4. Then create a new `Dog` saved in a variable `terrier`. Pass it two strings as arguments for the `name` and `color` properties.

Challenge Seed

```
function Dog() {

}
```

Solution

```
function Dog (name, color) {
  this.numLegs = 4;
  this.name = name;
  this.color = color;
}

const terrier = new Dog();
```

8. Verify an Object's Constructor with instanceof

Description

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
```

If an object is created without using a constructor, `instanceof` will verify that it is not an instance of that constructor:

```
let canary = {
  name: "Mildred",
  color: "Yellow",
  numLegs: 2
```

```
};

canary instanceof Bird; // => false
```

Instructions

Create a new instance of the `House` constructor, calling it `myHouse` and passing a number of bedrooms. Then, use `instanceof` to verify that it is an instance of `House`.

Challenge Seed

```
/* jshint expr: true */

function House(numBedrooms) {
  this.numBedrooms = numBedrooms;
}

// Add your code below this line
```

Solution

```
function House(numBedrooms) {
  this.numBedrooms = numBedrooms;
}
const myHouse = new House(4);
console.log(myHouse instanceof House);
```

9. Understand Own Properties

Description

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" ]
```

Instructions

Add the `own` properties of `canary` to the array `ownProps`.

Challenge Seed

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

let canary = new Bird("Tweety");
let ownProps = [];
// Add your code below this line
```

Solution

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

let canary = new Bird("Tweety");
function getOwnProps (obj) {
  const props = [];

  for (let prop in obj) {
    if (obj.hasOwnProperty(prop)) {
      props.push(prop);
    }
  }

  return props;
}

const ownProps = getOwnProps(canary);
```

10. Use Prototype Properties to Reduce Duplicate Code

Description

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.

Instructions

Add a `numLegs` property to the `prototype` of `Dog`

Challenge Seed

```
function Dog(name) {
  this.name = name;
}



// Add your code above this line
let beagle = new Dog("Snoopy");
```

Solution

```
function Dog (name) {
  this.name = name;
}
Dog.prototype.numLegs = 4;
let beagle = new Dog("Snoopy");
```

11. Iterate Over All Properties

Description

You have now seen two kinds of properties: `own` properties and `prototype` properties. `Own` properties are defined directly on the object instance itself. And `prototype` properties are defined on the `prototype`.

```
function Bird(name) {
  this.name = name; //own property
}

Bird.prototype.numLegs = 2; // prototype property

let duck = new Bird("Donald");
```

Here is how you add `duck`'s `own` properties to the array `ownProps` and `prototype` properties to the array `prototypeProps`:

```
let ownProps = [];
let prototypeProps = [];

for (let property in duck) {
  if(duck.hasOwnProperty(property)) {
    ownProps.push(property);
  } else {
    prototypeProps.push(property);
  }
}

console.log(ownProps); // prints ["name"]
console.log(prototypeProps); // prints ["numLegs"]
```

Instructions

Add all of the `own` properties of `beagle` to the array `ownProps`. Add all of the `prototype` properties of `Dog` to the array `prototypeProps`.

Challenge Seed

```
function Dog(name) {
  this.name = name;
```

```
}

Dog.prototype.numLegs = 4;

let beagle = new Dog("Snoopy");

let ownProps = [];
let prototypeProps = [];

// Add your code below this line
```

Solution

```
function Dog(name) {
  this.name = name;
}

Dog.prototype.numLegs = 4;

let beagle = new Dog("Snoopy");

let ownProps = [];
let prototypeProps = [];
for (let prop in beagle) {
  if (beagle.hasOwnProperty(prop)) {
    ownProps.push(prop);
  } else {
    prototypeProps.push(prop);
  }
}
```

12. Understand the Constructor Property

Description

There is a special `constructor` property located on the object instances `duck` and `beagle` that were created in the previous challenges:

```
let duck = new Bird();
let beagle = new Dog();

console.log(duck.constructor === Bird); //prints true
console.log(beagle.constructor === Dog); //prints true
```

Note that the `constructor` property is a reference to the constructor function that created the instance. The advantage of the `constructor` property is that it's possible to check for this property to find out what kind of object it is. Here's an example of how this could be used:

```
function joinBirdFraternity(candidate) {
  if (candidate.constructor === Bird) {
    return true;
  } else {
    return false;
  }
}
```

Note
Since the `constructor` property can be overwritten (which will be covered in the next two challenges) it's generally better to use the `instanceof` method to check the type of an object.

Instructions

Write a `joinDogFraternity` function that takes a `candidate` parameter and, using the `constructor` property, return `true` if the candidate is a `Dog`, otherwise return `false`.

Challenge Seed

```
function Dog(name) {
  this.name = name;
}

// Add your code below this line
function joinDogFraternity(candidate) {

}
```

Solution

```
function Dog(name) {
  this.name = name;
}
function joinDogFraternity(candidate) {
  return candidate.constructor === Dog;
}
```

13. Change the Prototype to a New Object

Description

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.

> **Bird.prototype.eat = function() {**
> **console.log("nom nom nom");**
> **}**
>
> **Bird.prototype.describe = function() {**
> **console.log("My name is " + this.name);**
> **}**

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:

> **Bird.prototype = {**
> **numLegs: 2,**
> **eat: function() {**
> **console.log("nom nom nom");**
> **},**
> **describe: function() {**
> **console.log("My name is " + this.name);**
> **}**
> **};**

Instructions

Add the property `numLegs` and the two methods `eat()` and `describe()` to the `prototype` of `Dog` by setting the `prototype` to a new object.

Challenge Seed

```
function Dog(name) {
  this.name = name;
}

Dog.prototype = {
  // Add your code below this line

};
```

Solution

```
function Dog(name) {
  this.name = name;
}
Dog.prototype = {
numLegs: 4,
  eat () {
    console.log('nom nom nom');
  },
  describe () {
    console.log('My name is ' + this.name);
  }
};
```

14. Remember to Set the Constructor Property when Changing the Prototype

Description

There is one crucial side effect of manually setting the prototype to a new object. It erases the `constructor` property! This property can be used to check which constructor function created the instance, but since the property has been overwritten, it now gives false results:

> **duck.constructor === Bird; // false -- Oops**
> **duck.constructor === Object; // true, all objects inherit from Object.prototype**
> **duck instanceof Bird; // true, still works**

To fix this, 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);
 }
};
```

Instructions

Define the `constructor` property on the `Dog prototype` .

Challenge Seed

```
function Dog(name) {
  this.name = name;
}

// Modify the code below this line
```

```
Dog.prototype = {

  numLegs: 4,
  eat: function() {
    console.log("nom nom nom");
  },
  describe: function() {
    console.log("My name is " + this.name);
  }
};
```

Solution

```
function Dog(name) {
  this.name = name;
}
Dog.prototype = {
  constructor: Dog,
  numLegs: 4,
  eat: function() {
    console.log("nom nom nom");
  },
  describe: function() {
    console.log("My name is " + this.name);
  }
};
```

15. Understand Where an Object's Prototype Comes From

Description

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
```

Instructions

Use `isPrototypeOf` to check the `prototype` of `beagle`.

Challenge Seed

```
function Dog(name) {
  this.name = name;
}

let beagle = new Dog("Snoopy");

// Add your code below this line
```

Solution

```
function Dog(name) {
  this.name = name;
}
let beagle = new Dog("Snoopy");
Dog.prototype.isPrototypeOf(beagle);
```

16. Understand the Prototype Chain

Description

All objects in JavaScript (with a few exceptions) have a `prototype`. Also, an object's `prototype` itself is an object.

```
function Bird(name) {
  this.name = name;
}

typeof Bird.prototype; // => object
```

Because a `prototype` is an object, a `prototype` can have its own `prototype`! In this case, the `prototype` of `Bird.prototype` is `Object.prototype`:

```
Object.prototype.isPrototypeOf(Bird.prototype);
// returns true
```

How is this useful? You may recall the `hasOwnProperty` method from a previous challenge:

```
let duck = new Bird("Donald");
duck.hasOwnProperty("name"); // => true
```

The `hasOwnProperty` method is defined in `Object.prototype`, which can be accessed by `Bird.prototype`, which can then be accessed by `duck`. This is an example of the `prototype` chain. In this `prototype` chain, `Bird` is the `supertype` for `duck`, while `duck` is the `subtype`. `Object` is a `supertype` for both `Bird` and `duck`. `Object` is a `supertype` for all objects in JavaScript. Therefore, any object can use the `hasOwnProperty` method.

Instructions

Modify the code to show the correct prototype chain.

Challenge Seed

```
function Dog(name) {
  this.name = name;
}

let beagle = new Dog("Snoopy");

Dog.prototype.isPrototypeOf(beagle);  // => true

// Fix the code below so that it evaluates to true
???.isPrototypeOf(Dog.prototype);
```

Solution

```
function Dog(name) {
  this.name = name;
}
let beagle = new Dog("Snoopy");
Dog.prototype.isPrototypeOf(beagle);
Object.prototype.isPrototypeOf(Dog.prototype);
```

17. Use Inheritance So You Don't Repeat Yourself

Description

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. Notice in the example below that the `describe` method is shared by `Bird` and `Dog`:

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

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

The `describe` method is repeated in two places. 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`:

```
Bird.prototype = {
  constructor: Bird
};

Dog.prototype = {
  constructor: Dog
};
```

Instructions

The `eat` method is repeated in both `Cat` and `Bear`. Edit the code in the spirit of `DRY` by moving the `eat` method to the `Animal supertype`.

Challenge Seed

```
function Cat(name) {
  this.name = name;
}

Cat.prototype = {
  constructor: Cat,
  eat: function() {
    console.log("nom nom nom");
  }
};

function Bear(name) {
  this.name = name;
}
```

```
Bear.prototype = {
  constructor: Bear,
  eat: function() {
    console.log("nom nom nom");
  }
};

function Animal() { }

Animal.prototype = {
  constructor: Animal,

};
```

Solution

```
function Cat(name) {
  this.name = name;
}

Cat.prototype = {
  constructor: Cat
};

function Bear(name) {
  this.name = name;
}

Bear.prototype = {
  constructor: Bear
};

function Animal() { }

Animal.prototype = {
  constructor: Animal,
  eat: function() {
    console.log("nom nom nom");
  }
};
```

18. Inherit Behaviors from a Supertype

Description

In the previous challenge, you created a `supertype` called `Animal` that defined behaviors shared by all animals:

> **function Animal() { }**
> **Animal.prototype.eat = function() {**
> **console.log("nom nom nom");**
> **};**

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's prototype`, you are effectively giving the `animal` instance the same "recipe" as any other instance of `Animal`.

> **animal.eat(); // prints "nom nom nom"**
> **animal instanceof Animal; // => true**

Instructions

Use `Object.create` **to make two instances of** `Animal` **named** `duck` **and** `beagle`**.**

Challenge Seed

```
function Animal() { }

Animal.prototype = {
  constructor: Animal,
  eat: function() {
    console.log("nom nom nom");
  }
};

// Add your code below this line

let duck; // Change this line
let beagle; // Change this line

duck.eat(); // Should print "nom nom nom"
beagle.eat(); // Should print "nom nom nom"
```

Solution

```
function Animal() { }

Animal.prototype = {
  constructor: Animal,
  eat: function() {
    console.log("nom nom nom");
  }
};
let duck = Object.create(Animal.prototype);
let beagle = Object.create(Animal.prototype);

duck.eat();
beagle.eat();
```

19. Set the Child's Prototype to an Instance of the Parent

Description

In the previous challenge you saw the first step for inheriting behavior from the `supertype` **(or parent)** `Animal`**: making a new instance of** `Animal`**. 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** `Bird` **now 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.**

Instructions

Modify the code so that instances of `Dog` **inherit from** `Animal`**.**

Challenge Seed

```
function Animal() { }

Animal.prototype = {
  constructor: Animal,
  eat: function() {
    console.log("nom nom nom");
  }
};

function Dog() { }

// Add your code below this line


let beagle = new Dog();
beagle.eat();  // Should print "nom nom nom"
```

Solution

```
function Animal() { }

Animal.prototype = {
  constructor: Animal,
  eat: function() {
    console.log("nom nom nom");
  }
};

function Dog() { }
Dog.prototype = Object.create(Animal.prototype);

let beagle = new Dog();
beagle.eat();
```

20. Reset an Inherited Constructor Property

Description

When an object inherits its `prototype` from another object, it also inherits the `supertype` 's constructor property. Here's an example:

> **function Bird() { }**
> **Bird.prototype = Object.create(Animal.prototype);**
> **let duck = new Bird();**
> **duck.constructor // function Animal(){...}**

But `duck` and all instances of `Bird` should 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(){...}**

Instructions

Fix the code so `duck.constructor` and `beagle.constructor` return their respective constructors.

Challenge Seed

```
function Animal() { }
function Bird() { }
function Dog() { }
```

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

// Add your code below this line



let duck = new Bird();
let beagle = new Dog();
```

Solution

```
function Animal() { }
function Bird() { }
function Dog() { }
Bird.prototype = Object.create(Animal.prototype);
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;
Bird.prototype.constructor = Bird;
let duck = new Bird();
let beagle = new Dog();
```

21. Add Methods After Inheritance

Description

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 `Bird` objects. Here, `Bird` will get a `fly()` function. Functions are added to `Bird's prototype` the 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!"
```

Instructions

Add all necessary code so the `Dog` object inherits from `Animal` and the `Dog's prototype` constructor is set to Dog. Then add a `bark()` method to the `Dog` object so that `beagle` can both `eat()` and `bark()`. The `bark()` method should print "Woof!" to the console.

Challenge Seed

```
function Animal() { }
Animal.prototype.eat = function() { console.log("nom nom nom"); };
```

```
function Dog() { }

// Add your code below this line



// Add your code above this line

let beagle = new Dog();

beagle.eat(); // Should print "nom nom nom"
beagle.bark(); // Should print "Woof!"
```

Solution

```
function Animal() { }
Animal.prototype.eat = function() { console.log("nom nom nom"); };

function Dog() { }
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;
Dog.prototype.bark = function () {
  console.log('Woof!');
};
let beagle = new Dog();

beagle.eat();
beagle.bark();
```

22. Override Inherited Methods

Description

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 `ChildObject` received 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.prototype` using the same method name as the one to override. Here's an example of `Bird` overriding 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";
};
```

If you have an instance `let duck = new Bird();` and you call `duck.eat()`, this is how JavaScript looks for the method on `duck's prototype` chain: 1. duck => Is eat() defined here? No. 2. Bird => Is eat() defined here? => Yes. Execute it and stop searching. 3. Animal => eat() is also defined, but JavaScript stopped searching before reaching this level. 4. Object => JavaScript stopped searching before reaching this level.

Instructions

Override the `fly()` method for `Penguin` so that it returns "Alas, this is a flightless bird."

Challenge Seed

```
function Bird() { }

Bird.prototype.fly = function() { return "I am flying!"; };

function Penguin() { }
Penguin.prototype = Object.create(Bird.prototype);
Penguin.prototype.constructor = Penguin;

// Add your code below this line



// Add your code above this line

let penguin = new Penguin();
console.log(penguin.fly());
```

Solution

```
function Bird() { }

Bird.prototype.fly = function() { return "I am flying!"; };

function Penguin() { }
Penguin.prototype = Object.create(Bird.prototype);
Penguin.prototype.constructor = Penguin;
Penguin.prototype.fly = () => 'Alas, this is a flightless bird.';
let penguin = new Penguin();
console.log(penguin.fly());
```

23. Use a Mixin to Add Common Behavior Between Unrelated Objects

Description

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 `bird` and `plane` are 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`.

Instructions

Create a `mixin` named `glideMixin` that defines a method named `glide`. Then use the `glideMixin` to give both `bird` and `boat` the ability to glide.

Challenge Seed

```
let bird = {
  name: "Donald",
  numLegs: 2
};

let boat = {
  name: "Warrior",
  type: "race-boat"
};

// Add your code below this line
```

Solution

```
let bird = {
  name: "Donald",
  numLegs: 2
};

let boat = {
  name: "Warrior",
  type: "race-boat"
};
function glideMixin (obj) {
  obj.glide = () => 'Gliding!';
}

glideMixin(bird);
glideMixin(boat);
```

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

Description

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 this public property 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 variable can only be accessed and changed by methods also within the constructor function.

```
function Bird() {
  let hatchedEgg = 10; // private variable

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

Here `getHachedEggCount` is 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`.

Instructions

Change how `weight` is declared in the `Bird` function so it is a private variable. Then, create a method `getWeight` that returns the value of `weight` 15.

Challenge Seed

```
function Bird() {
  this.weight = 15;


}
```

Solution

```
function Bird() {
  let weight = 15;

  this.getWeight = () => weight;
}
```

25. Understand the Immediately Invoked Function Expression (IIFE)

Description

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`.

Instructions

Rewrite the function `makeNest` and remove its call so instead it's an anonymous `immediately invoked function expression` (`IIFE`).

Challenge Seed

```
function makeNest() {
  console.log("A cozy nest is ready");
}

makeNest();
```

Solution

```
(function () {
  console.log("A cozy nest is ready");
})();
```

26. Use an IIFE to Create a Module

Description

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 `mixin` behaviors as properties of the object. The advantage of the `module` pattern 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();
```

Instructions

Create a `module` named `funModule` to wrap the two `mixins` `isCuteMixin` and `singMixin`. `funModule` should return an object.

Challenge Seed

```
let isCuteMixin = function(obj) {
  obj.isCute = function() {
    return true;
  };
};
let singMixin = function(obj) {
  obj.sing = function() {
    console.log("Singing to an awesome tune");
  };
};
```

Solution

```
const funModule = (function () {
  return {
    isCuteMixin: obj => {
      obj.isCute = () => true;
    },
    singMixin: obj => {
      obj.sing = () => console.log("Singing to an awesome tune");
    }
  };
})();
```

Functional Programming

1. Learn About Functional Programming

Description

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

Instructions

The members of freeCodeCamp happen to love tea. In the code editor, the `prepareTea` and `getTea` functions are already defined for you. Call the `getTea` function to get 40 cups of tea for the team, and store them in the `tea4TeamFCC` variable.

Challenge Seed

```
/**
 * A long process to prepare tea.
 * @return {string} A cup of tea.
 **/
const prepareTea = () => 'greenTea';

/**
 * Get given number of cups of tea.
 * @param {number} numOfCups Number of required cups of tea.
 * @return {Array<string>} Given amount of tea cups.
 **/
const getTea = (numOfCups) => {
  const teaCups = [];

  for(let cups = 1; cups <= numOfCups; cups += 1) {
    const teaCup = prepareTea();
    teaCups.push(teaCup);
  }

  return teaCups;
};

// Add your code below this line

const tea4TeamFCC = null; // :(

// Add your code above this line

console.log(tea4TeamFCC);
```

Solution

```
// solution required
```

2. Understand Functional Programming Terminology

Description

The FCC Team had a mood swing and now wants two types of tea: green tea and black tea. General Fact: Client mood swings are pretty common. With that information, we'll need to revisit the `getTea` function from last challenge to handle various tea requests. We can modify `getTea` to accept a function as a parameter to be able to change the type of tea it prepares. This makes `getTea` more flexible, and gives the programmer more control when client requests change. But first, let's cover some functional 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`.

Instructions

Prepare 27 cups of green tea and 13 cups of black tea and store them in `tea4GreenTeamFCC` and `tea4BlackTeamFCC` variables, respectively. Note that the `getTea` function has been modified so it now takes a function as the first argument. Note: The data (the number of cups of tea) is supplied as the last argument. We'll discuss this more in later lessons.

Challenge Seed

```
/**
 * A long process to prepare green tea.
 * @return {string} A cup of green tea.
 **/
const prepareGreenTea = () => 'greenTea';

/**
 * A long process to prepare black tea.
 * @return {string} A cup of black tea.
 **/
const prepareBlackTea = () => 'blackTea';

/**
 * Get given number of cups of tea.
 * @param {function():string} prepareTea The type of tea preparing function.
 * @param {number} numOfCups Number of required cups of tea.
 * @return {Array<string>} Given amount of tea cups.
 **/
const getTea = (prepareTea, numOfCups) => {
  const teaCups = [];

  for(let cups = 1; cups <= numOfCups; cups += 1) {
    const teaCup = prepareTea();
    teaCups.push(teaCup);
  }

  return teaCups;
};

// Add your code below this line

const tea4GreenTeamFCC = null; // :(
const tea4BlackTeamFCC = null; // :(

// Add your code above this line

console.log(
  tea4GreenTeamFCC,
  tea4BlackTeamFCC
);
```

Solution

```
// solution required
```

3. Understand the Hazards of Using Imperative Code

Description

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. 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. Consider the scenario: you are browsing the web in your browser, and want to track the tabs you have opened. Let's try to model this using some simple object-oriented code. A Window object is made up of tabs, and you usually have more than one Window open. The titles of each open site in each Window object is held in an array. After working in the browser (opening new tabs, merging windows, and closing tabs), you want to print the tabs that are still open. Closed tabs are removed from

the array and new tabs (for simplicity) get added to the end of it. The code editor shows an implementation of this functionality with functions for `tabOpen()`, `tabClose()`, and `join()`. The array `tabs` is part of the Window object that stores the name of the open pages.

Instructions

Run the code in the editor. It's using a method that has side effects in the program, causing incorrect output. The final list of open tabs should be `['FB', 'Gitter', 'Reddit', 'Twitter', 'Medium', 'new tab', 'Netflix', 'YouTube', 'Vine', 'GMail', 'Work mail', 'Docs', 'freeCodeCamp', 'new tab']` but the output will be slightly different. Work through the code and see if you can figure out the problem, then advance to the next challenge to learn more.

Instructions

Challenge Seed

```
// tabs is an array of titles of each site open within the window
var Window = function(tabs) {
  this.tabs = tabs; // we keep a record of the array inside the object
};

// When you join two windows into one window
Window.prototype.join = function (otherWindow) {
  this.tabs = this.tabs.concat(otherWindow.tabs);
  return this;
};

// When you open a new tab at the end
Window.prototype.tabOpen = function (tab) {
  this.tabs.push('new tab'); // let's open a new tab for now
  return this;
};

// When you close a tab
Window.prototype.tabClose = function (index) {
  var tabsBeforeIndex = this.tabs.splice(0, index); // get the tabs before the tab
  var tabsAfterIndex = this.tabs.splice(index); // get the tabs after the tab

  this.tabs = tabsBeforeIndex.concat(tabsAfterIndex); // join them together
  return this;
 };

// Let's create three browser windows
var workWindow = new Window(['GMail', 'Inbox', 'Work mail', 'Docs', 'freeCodeCamp']); // Your mailbox,
drive, and other work sites
var socialWindow = new Window(['FB', 'Gitter', 'Reddit', 'Twitter', 'Medium']); // Social sites
var videoWindow = new Window(['Netflix', 'YouTube', 'Vimeo', 'Vine']); //  Entertainment sites

// Now perform the tab opening, closing, and other operations
var finalTabs = socialWindow
                    .tabOpen() // Open a new tab for cat memes
                    .join(videoWindow.tabClose(2)) // Close third tab in video window, and join
                    .join(workWindow.tabClose(1).tabOpen());

alert(finalTabs.tabs);
```

Solution

```
// solution required
```

4. Avoid Mutations and Side Effects Using Functional Programming

Description

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 principles 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. Let's try to master this discipline and not alter any variable or object in our code.

Instructions

Fill in the code for the function `incrementer` so it returns the value of the global variable `fixedValue` increased by one.

Challenge Seed

```
// the global variable
var fixedValue = 4;

function incrementer () {
  // Add your code below this line


  // Add your code above this line
}

var newValue = incrementer(); // Should equal 5
console.log(fixedValue); // Should print 4
```

Solution

```
var fixedValue = 4

function incrementer() {
  return fixedValue + 1
}

var newValue = incrementer(); // Should equal 5
```

5. Pass Arguments to Avoid External Dependence in a Function

Description

The last challenge was a step closer to functional programming principles, but there is still something missing. We didn't alter the global variable value, but the function `incrementer` would not work without the global variable `fixedValue` being there. 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.

Instructions

Let's update the `incrementer` function to clearly declare its dependencies. Write the `incrementer` function so it takes an argument, and then increases the value by one.

Challenge Seed

```
// the global variable
var fixedValue = 4;

// Add your code below this line
function incrementer () {


  // Add your code above this line
}

var newValue = incrementer(fixedValue); // Should equal 5
console.log(fixedValue); // Should print 4
```

Solution

```
// the global variable
var fixedValue = 4;

const incrementer = val => val + 1;

var newValue = incrementer(fixedValue); // Should equal 5
console.log(fixedValue); // Should print 4
```

6. Refactor Global Variables Out of Functions

Description

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. Adding one to a number is not very exciting, but we can apply these principles when working with arrays or more complex objects.

Instructions

Rewrite the code so the global array `bookList` is not changed inside either function. The `add` function should add the given `bookName` to the end of an array. The `remove` function should remove the given `bookName` from an array. Both functions should return an array, and any new parameters should be added before the `bookName` parameter.

Challenge Seed

```
// the global variable
var bookList = ["The Hound of the Baskervilles", "On The Electrodynamics of Moving Bodies", "Philosophiæ
Naturalis Principia Mathematica", "Disquisitiones Arithmeticae"];

/* This function should add a book to the list and return the list */
// New parameters should come before bookName

// Add your code below this line
function add (bookName) {

  bookList.push(bookName);
```

```
  return bookList;

  // Add your code above this line
}

/* This function should remove a book from the list and return the list */
// New parameters should come before the bookName one

// Add your code below this line
function remove (bookName) {
  var book_index = bookList.indexOf(bookName);
  if (book_index >= 0) {

    bookList.splice(book_index, 1);
    return bookList;

    // Add your code above this line
    }
}

var newBookList = add(bookList, 'A Brief History of Time');
var newerBookList = remove(bookList, 'On The Electrodynamics of Moving Bodies');
var newestBookList = remove(add(bookList, 'A Brief History of Time'), 'On The Electrodynamics of Moving
Bodies');

console.log(bookList);
```

Solution

```
// solution required
```

7. Use the map Method to Extract Data from an Array

Description

So far we have learned to use pure functions to avoid side effects in a program. Also, we have seen the value in having a function only depend on its input arguments. This is only the beginning. As its name suggests, functional programming is centered around a theory of functions. 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.

Instructions

The `watchList` array holds objects with information on several movies. Use `map` to pull the title and rating from `watchList` and save the new array in the `rating` variable. The code in the editor currently uses a `for` loop to do this, replace the loop functionality with your `map` expression.

Challenge Seed

```
// the global variable
var watchList = [
                {
                  "Title": "Inception",
                  "Year": "2010",
                  "Rated": "PG-13",
                  "Released": "16 Jul 2010",
                  "Runtime": "148 min",
                  "Genre": "Action, Adventure, Crime",
                  "Director": "Christopher Nolan",
```

```
                    "Writer": "Christopher Nolan",
                    "Actors": "Leonardo DiCaprio, Joseph Gordon-Levitt, Ellen Page, Tom Hardy",
                    "Plot": "A thief, who steals corporate secrets through use of dream-sharing
technology, is given the inverse task of planting an idea into the mind of a CEO.",
                    "Language": "English, Japanese, French",
                    "Country": "USA, UK",
                    "Awards": "Won 4 Oscars. Another 143 wins & 198 nominations.",
                    "Poster": "http://ia.media-
imdb.com/images/M/MV5BMjAxMzY3NjcxNF5BMl5BanBnXkFtZTcwNTI5OTM0Mw@@._V1_SX300.jpg",
                    "Metascore": "74",
                    "imdbRating": "8.8",
                    "imdbVotes": "1,446,708",
                    "imdbID": "tt1375666",
                    "Type": "movie",
                    "Response": "True"
                },
                {
                    "Title": "Interstellar",
                    "Year": "2014",
                    "Rated": "PG-13",
                    "Released": "07 Nov 2014",
                    "Runtime": "169 min",
                    "Genre": "Adventure, Drama, Sci-Fi",
                    "Director": "Christopher Nolan",
                    "Writer": "Jonathan Nolan, Christopher Nolan",
                    "Actors": "Ellen Burstyn, Matthew McConaughey, Mackenzie Foy, John Lithgow",
                    "Plot": "A team of explorers travel through a wormhole in space in an attempt to
ensure humanity's survival.",
                    "Language": "English",
                    "Country": "USA, UK",
                    "Awards": "Won 1 Oscar. Another 39 wins & 132 nominations.",
                    "Poster": "http://ia.media-
imdb.com/images/M/MV5BMjIxNTU4MzY4MF5BMl5BanBnXkFtZTgwMzM4ODI3MjE@._V1_SX300.jpg",
                    "Metascore": "74",
                    "imdbRating": "8.6",
                    "imdbVotes": "910,366",
                    "imdbID": "tt0816692",
                    "Type": "movie",
                    "Response": "True"
                },
                {
                    "Title": "The Dark Knight",
                    "Year": "2008",
                    "Rated": "PG-13",
                    "Released": "18 Jul 2008",
                    "Runtime": "152 min",
                    "Genre": "Action, Adventure, Crime",
                    "Director": "Christopher Nolan",
                    "Writer": "Jonathan Nolan (screenplay), Christopher Nolan (screenplay), Christopher
Nolan (story), David S. Goyer (story), Bob Kane (characters)",
                    "Actors": "Christian Bale, Heath Ledger, Aaron Eckhart, Michael Caine",
                    "Plot": "When the menace known as the Joker wreaks havoc and chaos on the people of
Gotham, the caped crusader must come to terms with one of the greatest psychological tests of his
ability to fight injustice.",
                    "Language": "English, Mandarin",
                    "Country": "USA, UK",
                    "Awards": "Won 2 Oscars. Another 146 wins & 142 nominations.",
                    "Poster": "http://ia.media-
imdb.com/images/M/MV5BMTMxNTMwODM0NF5BMl5BanBnXkFtZTcwODAyMTk2Mw@@._V1_SX300.jpg",
                    "Metascore": "82",
                    "imdbRating": "9.0",
                    "imdbVotes": "1,652,832",
                    "imdbID": "tt0468569",
                    "Type": "movie",
                    "Response": "True"
                },
                {
                    "Title": "Batman Begins",
                    "Year": "2005",
                    "Rated": "PG-13",
                    "Released": "15 Jun 2005",
                    "Runtime": "140 min",
                    "Genre": "Action, Adventure",
                    "Director": "Christopher Nolan",
                    "Writer": "Bob Kane (characters), David S. Goyer (story), Christopher Nolan
(screenplay), David S. Goyer (screenplay)",
                    "Actors": "Christian Bale, Michael Caine, Liam Neeson, Katie Holmes",
```

```
                    "Plot": "After training with his mentor, Batman begins his fight to free crime-ridden
Gotham City from the corruption that Scarecrow and the League of Shadows have cast upon it.",
                    "Language": "English, Urdu, Mandarin",
                    "Country": "USA, UK",
                    "Awards": "Nominated for 1 Oscar. Another 15 wins & 66 nominations.",
                    "Poster": "http://ia.media-
imdb.com/images/M/MV5BNTM3OTc0MzM2OV5BMl5BanBnXkFtZTYwNzUwMTI3._V1_SX300.jpg",
                    "Metascore": "70",
                    "imdbRating": "8.3",
                    "imdbVotes": "972,584",
                    "imdbID": "tt0372784",
                    "Type": "movie",
                    "Response": "True"
                },
                {
                    "Title": "Avatar",
                    "Year": "2009",
                    "Rated": "PG-13",
                    "Released": "18 Dec 2009",
                    "Runtime": "162 min",
                    "Genre": "Action, Adventure, Fantasy",
                    "Director": "James Cameron",
                    "Writer": "James Cameron",
                    "Actors": "Sam Worthington, Zoe Saldana, Sigourney Weaver, Stephen Lang",
                    "Plot": "A paraplegic marine dispatched to the moon Pandora on a unique mission
becomes torn between following his orders and protecting the world he feels is his home.",
                    "Language": "English, Spanish",
                    "Country": "USA, UK",
                    "Awards": "Won 3 Oscars. Another 80 wins & 121 nominations.",
                    "Poster": "http://ia.media-
imdb.com/images/M/MV5BMTYwOTEwNjAzMl5BMl5BanBnXkFtZTcwODc5MTUwMw@@._V1_SX300.jpg",
                    "Metascore": "83",
                    "imdbRating": "7.9",
                    "imdbVotes": "876,575",
                    "imdbID": "tt0499549",
                    "Type": "movie",
                    "Response": "True"
                }
];

// Add your code below this line

var rating = [];
for(var i=0; i < watchList.length; i++){
  rating.push({title: watchList[i]["Title"],  rating: watchList[i]["imdbRating"]});
}

// Add your code above this line

console.log(JSON.stringify(rating));
```

Solution

```
// the global variable
var watchList = [
                 {
                    "Title": "Inception",
                    "Year": "2010",
                    "Rated": "PG-13",
                    "Released": "16 Jul 2010",
                    "Runtime": "148 min",
                    "Genre": "Action, Adventure, Crime",
                    "Director": "Christopher Nolan",
                    "Writer": "Christopher Nolan",
                    "Actors": "Leonardo DiCaprio, Joseph Gordon-Levitt, Ellen Page, Tom Hardy",
                    "Plot": "A thief, who steals corporate secrets through use of dream-sharing
technology, is given the inverse task of planting an idea into the mind of a CEO.",
                    "Language": "English, Japanese, French",
                    "Country": "USA, UK",
                    "Awards": "Won 4 Oscars. Another 143 wins & 198 nominations.",
                    "Poster": "http://ia.media-
imdb.com/images/M/MV5BMjAxMzY3NjcxNF5BMl5BanBnXkFtZTcwNTI5OTM0Mw@@._V1_SX300.jpg",
                    "Metascore": "74",
                    "imdbRating": "8.8",
```

```
                    "imdbVotes": "1,446,708",
                    "imdbID": "tt1375666",
                    "Type": "movie",
                    "Response": "True"
                },
                {
                    "Title": "Interstellar",
                    "Year": "2014",
                    "Rated": "PG-13",
                    "Released": "07 Nov 2014",
                    "Runtime": "169 min",
                    "Genre": "Adventure, Drama, Sci-Fi",
                    "Director": "Christopher Nolan",
                    "Writer": "Jonathan Nolan, Christopher Nolan",
                    "Actors": "Ellen Burstyn, Matthew McConaughey, Mackenzie Foy, John Lithgow",
                    "Plot": "A team of explorers travel through a wormhole in space in an attempt to
ensure humanity's survival.",
                    "Language": "English",
                    "Country": "USA, UK",
                    "Awards": "Won 1 Oscar. Another 39 wins & 132 nominations.",
                    "Poster": "http://ia.media-
imdb.com/images/M/MV5BMjIxNTU4MzY4MF5BMl5BanBnXkFtZTgwMzM4ODI3MjE@._V1_SX300.jpg",
                    "Metascore": "74",
                    "imdbRating": "8.6",
                    "imdbVotes": "910,366",
                    "imdbID": "tt0816692",
                    "Type": "movie",
                    "Response": "True"
                },
                {
                    "Title": "The Dark Knight",
                    "Year": "2008",
                    "Rated": "PG-13",
                    "Released": "18 Jul 2008",
                    "Runtime": "152 min",
                    "Genre": "Action, Adventure, Crime",
                    "Director": "Christopher Nolan",
                    "Writer": "Jonathan Nolan (screenplay), Christopher Nolan (screenplay), Christopher
Nolan (story), David S. Goyer (story), Bob Kane (characters)",
                    "Actors": "Christian Bale, Heath Ledger, Aaron Eckhart, Michael Caine",
                    "Plot": "When the menace known as the Joker wreaks havoc and chaos on the people of
Gotham, the caped crusader must come to terms with one of the greatest psychological tests of his
ability to fight injustice.",
                    "Language": "English, Mandarin",
                    "Country": "USA, UK",
                    "Awards": "Won 2 Oscars. Another 146 wins & 142 nominations.",
                    "Poster": "http://ia.media-
imdb.com/images/M/MV5BMTMxNTMwODM0NF5BMl5BanBnXkFtZTcwODAyMTk2Mw@@._V1_SX300.jpg",
                    "Metascore": "82",
                    "imdbRating": "9.0",
                    "imdbVotes": "1,652,832",
                    "imdbID": "tt0468569",
                    "Type": "movie",
                    "Response": "True"
                },
                {
                    "Title": "Batman Begins",
                    "Year": "2005",
                    "Rated": "PG-13",
                    "Released": "15 Jun 2005",
                    "Runtime": "140 min",
                    "Genre": "Action, Adventure",
                    "Director": "Christopher Nolan",
                    "Writer": "Bob Kane (characters), David S. Goyer (story), Christopher Nolan
(screenplay), David S. Goyer (screenplay)",
                    "Actors": "Christian Bale, Michael Caine, Liam Neeson, Katie Holmes",
                    "Plot": "After training with his mentor, Batman begins his fight to free crime-ridden
Gotham City from the corruption that Scarecrow and the League of Shadows have cast upon it.",
                    "Language": "English, Urdu, Mandarin",
                    "Country": "USA, UK",
                    "Awards": "Nominated for 1 Oscar. Another 15 wins & 66 nominations.",
                    "Poster": "http://ia.media-
imdb.com/images/M/MV5BNTM3OTc0MzM2OV5BMl5BanBnXkFtZTYwNzUwMTI3._V1_SX300.jpg",
                    "Metascore": "70",
                    "imdbRating": "8.3",
                    "imdbVotes": "972,584",
                    "imdbID": "tt0372784",
```

```
                    "Type": "movie",
                    "Response": "True"
                },
                {
                    "Title": "Avatar",
                    "Year": "2009",
                    "Rated": "PG-13",
                    "Released": "18 Dec 2009",
                    "Runtime": "162 min",
                    "Genre": "Action, Adventure, Fantasy",
                    "Director": "James Cameron",
                    "Writer": "James Cameron",
                    "Actors": "Sam Worthington, Zoe Saldana, Sigourney Weaver, Stephen Lang",
                    "Plot": "A paraplegic marine dispatched to the moon Pandora on a unique mission
becomes torn between following his orders and protecting the world he feels is his home.",
                    "Language": "English, Spanish",
                    "Country": "USA, UK",
                    "Awards": "Won 3 Oscars. Another 80 wins & 121 nominations.",
                    "Poster": "http://ia.media-
imdb.com/images/M/MV5BMTYwOTEwNjAzMl5BMl5BanBnXkFtZTcwODc5MTUwMw@@._V1_SX300.jpg",
                    "Metascore": "83",
                    "imdbRating": "7.9",
                    "imdbVotes": "876,575",
                    "imdbID": "tt0499549",
                    "Type": "movie",
                    "Response": "True"
                }
];

var rating = watchList.map(function(movie) {
  return {
    title: movie["Title"],
    rating: movie["imdbRating"]
  }
});
```

8. Implement map on a Prototype

Description

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.

Instructions

Write your own `Array.prototype.myMap()`, which should behave exactly like `Array.prototype.map()`. You may use a `for` loop or the `forEach` method.

Challenge Seed

```
// the global Array
var s = [23, 65, 98, 5];

Array.prototype.myMap = function(callback){
  var newArray = [];
  // Add your code below this line

  // Add your code above this line
  return newArray;

};

var new_s = s.myMap(function(item){
```

```
  return item * 2;
});
```

Solution

```
// solution required
```

9. Use the filter Method to Extract Data from an Array

Description

Another useful array function is `Array.prototype.filter()`, or simply `filter()`. The `filter` method returns a new array which is at most as long as the original array, but usually has fewer items. `Filter` doesn't alter the original array, just like `map`. It takes a callback function that applies the logic inside the callback on each element of the array. If an element returns true based on the criteria in the callback function, then it is included in the new array.

Instructions

The variable `watchList` holds an array of objects with information on several movies. Use a combination of `filter` and `map` to return a new array of objects with only `title` and `rating` keys, but where `imdbRating` is greater than or equal to 8.0. Note that the rating values are saved as strings in the object and you may want to convert them into numbers to perform mathematical operations on them.

Challenge Seed

```
// the global variable
var watchList = [
                 {
                   "Title": "Inception",
                   "Year": "2010",
                   "Rated": "PG-13",
                   "Released": "16 Jul 2010",
                   "Runtime": "148 min",
                   "Genre": "Action, Adventure, Crime",
                   "Director": "Christopher Nolan",
                   "Writer": "Christopher Nolan",
                   "Actors": "Leonardo DiCaprio, Joseph Gordon-Levitt, Ellen Page, Tom Hardy",
                   "Plot": "A thief, who steals corporate secrets through use of dream-sharing
technology, is given the inverse task of planting an idea into the mind of a CEO.",
                   "Language": "English, Japanese, French",
                   "Country": "USA, UK",
                   "Awards": "Won 4 Oscars. Another 143 wins & 198 nominations.",
                   "Poster": "http://ia.media-
imdb.com/images/M/MV5BMjAxMzY3NjcxNF5BMl5BanBnXkFtZTcwNTI5OTM0Mw@@._V1_SX300.jpg",
                   "Metascore": "74",
                   "imdbRating": "8.8",
                   "imdbVotes": "1,446,708",
                   "imdbID": "tt1375666",
                   "Type": "movie",
                   "Response": "True"
                 },
                 {
                   "Title": "Interstellar",
                   "Year": "2014",
                   "Rated": "PG-13",
                   "Released": "07 Nov 2014",
                   "Runtime": "169 min",
                   "Genre": "Adventure, Drama, Sci-Fi",
                   "Director": "Christopher Nolan",
                   "Writer": "Jonathan Nolan, Christopher Nolan",
                   "Actors": "Ellen Burstyn, Matthew McConaughey, Mackenzie Foy, John Lithgow",
                   "Plot": "A team of explorers travel through a wormhole in space in an attempt to
ensure humanity's survival.",
```

```
                    "Language": "English",
                    "Country": "USA, UK",
                    "Awards": "Won 1 Oscar. Another 39 wins & 132 nominations.",
                    "Poster": "http://ia.media-
imdb.com/images/M/MV5BMjIxNTU4MzY4MF5BMl5BanBnXkFtZTgwMzM4ODI3MjE@._V1_SX300.jpg",
                    "Metascore": "74",
                    "imdbRating": "8.6",
                    "imdbVotes": "910,366",
                    "imdbID": "tt0816692",
                    "Type": "movie",
                    "Response": "True"
                },
                {
                    "Title": "The Dark Knight",
                    "Year": "2008",
                    "Rated": "PG-13",
                    "Released": "18 Jul 2008",
                    "Runtime": "152 min",
                    "Genre": "Action, Adventure, Crime",
                    "Director": "Christopher Nolan",
                    "Writer": "Jonathan Nolan (screenplay), Christopher Nolan (screenplay), Christopher
Nolan (story), David S. Goyer (story), Bob Kane (characters)",
                    "Actors": "Christian Bale, Heath Ledger, Aaron Eckhart, Michael Caine",
                    "Plot": "When the menace known as the Joker wreaks havoc and chaos on the people of
Gotham, the caped crusader must come to terms with one of the greatest psychological tests of his
ability to fight injustice.",
                    "Language": "English, Mandarin",
                    "Country": "USA, UK",
                    "Awards": "Won 2 Oscars. Another 146 wins & 142 nominations.",
                    "Poster": "http://ia.media-
imdb.com/images/M/MV5BMTMxNTMwODM0NF5BMl5BanBnXkFtZTcwODAyMTk2Mw@@._V1_SX300.jpg",
                    "Metascore": "82",
                    "imdbRating": "9.0",
                    "imdbVotes": "1,652,832",
                    "imdbID": "tt0468569",
                    "Type": "movie",
                    "Response": "True"
                },
                {
                    "Title": "Batman Begins",
                    "Year": "2005",
                    "Rated": "PG-13",
                    "Released": "15 Jun 2005",
                    "Runtime": "140 min",
                    "Genre": "Action, Adventure",
                    "Director": "Christopher Nolan",
                    "Writer": "Bob Kane (characters), David S. Goyer (story), Christopher Nolan
(screenplay), David S. Goyer (screenplay)",
                    "Actors": "Christian Bale, Michael Caine, Liam Neeson, Katie Holmes",
                    "Plot": "After training with his mentor, Batman begins his fight to free crime-ridden
Gotham City from the corruption that Scarecrow and the League of Shadows have cast upon it.",
                    "Language": "English, Urdu, Mandarin",
                    "Country": "USA, UK",
                    "Awards": "Nominated for 1 Oscar. Another 15 wins & 66 nominations.",
                    "Poster": "http://ia.media-
imdb.com/images/M/MV5BNTM3OTc0MzM2OV5BMl5BanBnXkFtZTYwNzUwMTI3._V1_SX300.jpg",
                    "Metascore": "70",
                    "imdbRating": "8.3",
                    "imdbVotes": "972,584",
                    "imdbID": "tt0372784",
                    "Type": "movie",
                    "Response": "True"
                },
                {
                    "Title": "Avatar",
                    "Year": "2009",
                    "Rated": "PG-13",
                    "Released": "18 Dec 2009",
                    "Runtime": "162 min",
                    "Genre": "Action, Adventure, Fantasy",
                    "Director": "James Cameron",
                    "Writer": "James Cameron",
                    "Actors": "Sam Worthington, Zoe Saldana, Sigourney Weaver, Stephen Lang",
                    "Plot": "A paraplegic marine dispatched to the moon Pandora on a unique mission
becomes torn between following his orders and protecting the world he feels is his home.",
                    "Language": "English, Spanish",
                    "Country": "USA, UK",
```

```
                    "Awards": "Won 3 Oscars. Another 80 wins & 121 nominations.",
                    "Poster": "http://ia.media-
imdb.com/images/M/MV5BMTYwOTEwNjAzMl5BMl5BanBnXkFtZTcwODc5MTUwMw@@._V1_SX300.jpg",
                    "Metascore": "83",
                    "imdbRating": "7.9",
                    "imdbVotes": "876,575",
                    "imdbID": "tt0499549",
                    "Type": "movie",
                    "Response": "True"
                }
];

// Add your code below this line

var filteredList;

// Add your code above this line

console.log(filteredList);
```

Solution

```
// solution required
```

10. Implement the filter Method on a Prototype

Description

It would teach us a lot about the `filter` method if we try to implement a version of it that behaves exactly like `Array.prototype.filter()`. It can use either 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.

Instructions

Write your own `Array.prototype.myFilter()`, which should behave exactly like `Array.prototype.filter()`. You may use a `for` loop or the `Array.prototype.forEach()` method.

Challenge Seed

```
// the global Array
var s = [23, 65, 98, 5];

Array.prototype.myFilter = function(callback){
  var newArray = [];
  // Add your code below this line

  // Add your code above this line
  return newArray;

};

var new_s = s.myFilter(function(item){
  return item % 2 === 1;
});
```

Solution

```
// solution required
```

11. Return Part of an Array Using the slice Method

Description

The `slice` method returns a copy of certain elements of an array. It can take two arguments, the first gives the index of where to begin the slice, the second is the index for where to end the slice (and it's non-inclusive). If the arguments are not provided, the default is to start at the beginning of the array through the end, which is an easy way to make a copy of the entire array. The `slice` method does not mutate the original array, but returns a new one. Here's an example:

```
var arr = ["Cat", "Dog", "Tiger", "Zebra"];
var newArray = arr.slice(1, 3);
// Sets newArray to ["Dog", "Tiger"]
```

Instructions

Use the `slice` method in the `sliceArray` function to return part of the `anim` array given the provided `beginSlice` and `endSlice` indices. The function should return an array.

Challenge Seed

```
function sliceArray(anim, beginSlice, endSlice) {
  // Add your code below this line


  // Add your code above this line
}
var inputAnim = ["Cat", "Dog", "Tiger", "Zebra", "Ant"];
sliceArray(inputAnim, 1, 3);
```

Solution

```
// solution required
```

12. Remove Elements from an Array Using slice Instead of splice

Description

A common pattern while working with arrays is when you want to remove items and keep the rest of the array. JavaScript offers the `splice` method for this, which takes arguments for the index of where to start removing items, then the number of items to remove. If the second argument is not provided, the default is to remove items through the end. However, the `splice` method mutates the original array it is called on. Here's an example:

```
var cities = ["Chicago", "Delhi", "Islamabad", "London", "Berlin"];
cities.splice(3, 1); // Returns "London" and deletes it from the cities array
// cities is now ["Chicago", "Delhi", "Islamabad", "Berlin"]
```

As we saw in the last challenge, the `slice` method does not mutate the original array, but returns a new one which can be saved into a variable. Recall that the `slice` method takes two arguments for the indices to begin and end the slice (the end is non-inclusive), and returns those items in a new array. Using the `slice` method instead of `splice` helps to avoid any array-mutating side effects.

Instructions

Rewrite the function `nonMutatingSplice` by using `slice` instead of `splice`. It should limit the provided `cities` array to a length of 3, and return a new array with only the first three items. Do not mutate the original array provided to the function.

Challenge Seed

```
function nonMutatingSplice(cities) {
  // Add your code below this line
  return cities.splice(3);

  // Add your code above this line
}
var inputCities = ["Chicago", "Delhi", "Islamabad", "London", "Berlin"];
nonMutatingSplice(inputCities);
```

Solution

```
// solution required
```

13. Combine Two Arrays Using the concat Method

Description

`Concatenation` means to join items end to end. JavaScript offers the `concat` method for both strings and arrays that work in the same way. For arrays, the method is called on one, then another array is provided as the argument to `concat`, which is added to the end of the first array. It returns a new array and does not mutate either of the original arrays. Here's an example:

> **[1, 2, 3].concat([4, 5, 6]);**
> **// Returns a new array [1, 2, 3, 4, 5, 6]**

Instructions

Use the `concat` method in the `nonMutatingConcat` function to concatenate `attach` to the end of `original`. The function should return the concatenated array.

Challenge Seed

```
function nonMutatingConcat(original, attach) {
  // Add your code below this line


  // Add your code above this line
}
var first = [1, 2, 3];
var second = [4, 5];
nonMutatingConcat(first, second);
```

Solution

```
// solution required
```

14. Add Elements to the End of an Array Using concat Instead of push

Description

Functional programming is all about creating and using non-mutating functions. The last challenge introduced the `concat` method as a way to combine arrays into a new one without mutating the original arrays. Compare `concat` to the `push` method. `Push` adds an item to the end of the same array it is called on, which mutates that array. Here's an example:

```
var arr = [1, 2, 3];
arr.push([4, 5, 6]);
// arr is changed to [1, 2, 3, [4, 5, 6]]
// Not the functional programming way
```

`Concat` offers a way to add new items to the end of an array without any mutating side effects.

Instructions

Change the `nonMutatingPush` function so it uses `concat` to add `newItem` to the end of `original` instead of `push`. The function should return an array.

Challenge Seed

```
function nonMutatingPush(original, newItem) {
  // Add your code below this line
  return original.push(newItem);

  // Add your code above this line
}
var first = [1, 2, 3];
var second = [4, 5];
nonMutatingPush(first, second);
```

Solution

```
// solution required
```

15. Use the reduce Method to Analyze Data

Description

`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 `reduce` method. 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`. However, before we get there, let's practice using `reduce` first.

Instructions

The variable `watchList` holds an array of objects with information on several movies. Use `reduce` to find the average IMDB rating of the movies directed by Christopher Nolan. Recall from prior challenges how to `filter` data and `map` over it to pull what you need. You may need to create other variables, and return the average rating from `getRating` function. Note that the rating values are saved as strings in the object and need to be converted into numbers before they are used in any mathematical operations.

Challenge Seed

```
// the global variable
var watchList = [
                 {
                   "Title": "Inception",
                   "Year": "2010",
                   "Rated": "PG-13",
                   "Released": "16 Jul 2010",
                   "Runtime": "148 min",
                   "Genre": "Action, Adventure, Crime",
                   "Director": "Christopher Nolan",
                   "Writer": "Christopher Nolan",
                   "Actors": "Leonardo DiCaprio, Joseph Gordon-Levitt, Ellen Page, Tom Hardy",
                   "Plot": "A thief, who steals corporate secrets through use of dream-sharing
technology, is given the inverse task of planting an idea into the mind of a CEO.",
                   "Language": "English, Japanese, French",
                   "Country": "USA, UK",
                   "Awards": "Won 4 Oscars. Another 143 wins & 198 nominations.",
                   "Poster": "http://ia.media-
imdb.com/images/M/MV5BMjAxMzY3NjcxNF5BMl5BanBnXkFtZTcwNTI5OTM0Mw@@._V1_SX300.jpg",
                   "Metascore": "74",
                   "imdbRating": "8.8",
                   "imdbVotes": "1,446,708",
                   "imdbID": "tt1375666",
                   "Type": "movie",
                   "Response": "True"
               },
               {
                   "Title": "Interstellar",
                   "Year": "2014",
                   "Rated": "PG-13",
                   "Released": "07 Nov 2014",
                   "Runtime": "169 min",
                   "Genre": "Adventure, Drama, Sci-Fi",
                   "Director": "Christopher Nolan",
                   "Writer": "Jonathan Nolan, Christopher Nolan",
                   "Actors": "Ellen Burstyn, Matthew McConaughey, Mackenzie Foy, John Lithgow",
                   "Plot": "A team of explorers travel through a wormhole in space in an attempt to
ensure humanity's survival.",
                   "Language": "English",
                   "Country": "USA, UK",
                   "Awards": "Won 1 Oscar. Another 39 wins & 132 nominations.",
                   "Poster": "http://ia.media-
imdb.com/images/M/MV5BMjIxNTU4MzY4MF5BMl5BanBnXkFtZTgwMzM4ODI3MjE@._V1_SX300.jpg",
                   "Metascore": "74",
                   "imdbRating": "8.6",
                   "imdbVotes": "910,366",
                   "imdbID": "tt0816692",
                   "Type": "movie",
                   "Response": "True"
               },
               {
                   "Title": "The Dark Knight",
                   "Year": "2008",
                   "Rated": "PG-13",
                   "Released": "18 Jul 2008",
                   "Runtime": "152 min",
                   "Genre": "Action, Adventure, Crime",
                   "Director": "Christopher Nolan",
                   "Writer": "Jonathan Nolan (screenplay), Christopher Nolan (screenplay), Christopher
Nolan (story), David S. Goyer (story), Bob Kane (characters)",
                   "Actors": "Christian Bale, Heath Ledger, Aaron Eckhart, Michael Caine",
                   "Plot": "When the menace known as the Joker wreaks havoc and chaos on the people of
Gotham, the caped crusader must come to terms with one of the greatest psychological tests of his
ability to fight injustice.",
                   "Language": "English, Mandarin",
                   "Country": "USA, UK",
                   "Awards": "Won 2 Oscars. Another 146 wins & 142 nominations.",
                   "Poster": "http://ia.media-
imdb.com/images/M/MV5BMTMxNTMwODM0NF5BMl5BanBnXkFtZTcwODAyMTk2Mw@@._V1_SX300.jpg",
                   "Metascore": "82",
                   "imdbRating": "9.0",
                   "imdbVotes": "1,652,832",
                   "imdbID": "tt0468569",
                   "Type": "movie",
                   "Response": "True"
               },
```

```
                {
                   "Title": "Batman Begins",
                   "Year": "2005",
                   "Rated": "PG-13",
                   "Released": "15 Jun 2005",
                   "Runtime": "140 min",
                   "Genre": "Action, Adventure",
                   "Director": "Christopher Nolan",
                   "Writer": "Bob Kane (characters), David S. Goyer (story), Christopher Nolan
(screenplay), David S. Goyer (screenplay)",
                   "Actors": "Christian Bale, Michael Caine, Liam Neeson, Katie Holmes",
                   "Plot": "After training with his mentor, Batman begins his fight to free crime-ridden
Gotham City from the corruption that Scarecrow and the League of Shadows have cast upon it.",
                   "Language": "English, Urdu, Mandarin",
                   "Country": "USA, UK",
                   "Awards": "Nominated for 1 Oscar. Another 15 wins & 66 nominations.",
                   "Poster": "http://ia.media-
imdb.com/images/M/MV5BNTM3OTc0MzM2OV5BMl5BanBnXkFtZTYwNzUwMTI3._V1_SX300.jpg",
                   "Metascore": "70",
                   "imdbRating": "8.3",
                   "imdbVotes": "972,584",
                   "imdbID": "tt0372784",
                   "Type": "movie",
                   "Response": "True"
                },
                {
                   "Title": "Avatar",
                   "Year": "2009",
                   "Rated": "PG-13",
                   "Released": "18 Dec 2009",
                   "Runtime": "162 min",
                   "Genre": "Action, Adventure, Fantasy",
                   "Director": "James Cameron",
                   "Writer": "James Cameron",
                   "Actors": "Sam Worthington, Zoe Saldana, Sigourney Weaver, Stephen Lang",
                   "Plot": "A paraplegic marine dispatched to the moon Pandora on a unique mission
becomes torn between following his orders and protecting the world he feels is his home.",
                   "Language": "English, Spanish",
                   "Country": "USA, UK",
                   "Awards": "Won 3 Oscars. Another 80 wins & 121 nominations.",
                   "Poster": "http://ia.media-
imdb.com/images/M/MV5BMTYwOTEwNjAzMl5BMl5BanBnXkFtZTcwODc5MTUwMw@@._V1_SX300.jpg",
                   "Metascore": "83",
                   "imdbRating": "7.9",
                   "imdbVotes": "876,575",
                   "imdbID": "tt0499549",
                   "Type": "movie",
                   "Response": "True"
                }
];

function getRating(watchList){
  // Add your code below this line
  var averageRating;


  // Add your code above this line
  return averageRating;
}
console.log(getRating(watchList));
```

Solution

```
// the global variable
var watchList = [
                 {
                   "Title": "Inception",
                   "Year": "2010",
                   "Rated": "PG-13",
                   "Released": "16 Jul 2010",
                   "Runtime": "148 min",
                   "Genre": "Action, Adventure, Crime",
                   "Director": "Christopher Nolan",
                   "Writer": "Christopher Nolan",
```

```
                    "Actors": "Leonardo DiCaprio, Joseph Gordon-Levitt, Ellen Page, Tom Hardy",
                    "Plot": "A thief, who steals corporate secrets through use of dream-sharing
technology, is given the inverse task of planting an idea into the mind of a CEO.",
                    "Language": "English, Japanese, French",
                    "Country": "USA, UK",
                    "Awards": "Won 4 Oscars. Another 143 wins & 198 nominations.",
                    "Poster": "http://ia.media-
imdb.com/images/M/MV5BMjAxMzY3NjcxNF5BMl5BanBnXkFtZTcwNTI5OTM0Mw@@._V1_SX300.jpg",
                    "Metascore": "74",
                    "imdbRating": "8.8",
                    "imdbVotes": "1,446,708",
                    "imdbID": "tt1375666",
                    "Type": "movie",
                    "Response": "True"
                },
                {
                    "Title": "Interstellar",
                    "Year": "2014",
                    "Rated": "PG-13",
                    "Released": "07 Nov 2014",
                    "Runtime": "169 min",
                    "Genre": "Adventure, Drama, Sci-Fi",
                    "Director": "Christopher Nolan",
                    "Writer": "Jonathan Nolan, Christopher Nolan",
                    "Actors": "Ellen Burstyn, Matthew McConaughey, Mackenzie Foy, John Lithgow",
                    "Plot": "A team of explorers travel through a wormhole in space in an attempt to
ensure humanity's survival.",
                    "Language": "English",
                    "Country": "USA, UK",
                    "Awards": "Won 1 Oscar. Another 39 wins & 132 nominations.",
                    "Poster": "http://ia.media-
imdb.com/images/M/MV5BMjIxNTU4MzY4MF5BMl5BanBnXkFtZTgwMzM4ODI3MjE@._V1_SX300.jpg",
                    "Metascore": "74",
                    "imdbRating": "8.6",
                    "imdbVotes": "910,366",
                    "imdbID": "tt0816692",
                    "Type": "movie",
                    "Response": "True"
                },
                {
                    "Title": "The Dark Knight",
                    "Year": "2008",
                    "Rated": "PG-13",
                    "Released": "18 Jul 2008",
                    "Runtime": "152 min",
                    "Genre": "Action, Adventure, Crime",
                    "Director": "Christopher Nolan",
                    "Writer": "Jonathan Nolan (screenplay), Christopher Nolan (screenplay), Christopher
Nolan (story), David S. Goyer (story), Bob Kane (characters)",
                    "Actors": "Christian Bale, Heath Ledger, Aaron Eckhart, Michael Caine",
                    "Plot": "When the menace known as the Joker wreaks havoc and chaos on the people of
Gotham, the caped crusader must come to terms with one of the greatest psychological tests of his
ability to fight injustice.",
                    "Language": "English, Mandarin",
                    "Country": "USA, UK",
                    "Awards": "Won 2 Oscars. Another 146 wins & 142 nominations.",
                    "Poster": "http://ia.media-
imdb.com/images/M/MV5BMTMxNTMwODM0NF5BMl5BanBnXkFtZTcwODAyMTk2Mw@@._V1_SX300.jpg",
                    "Metascore": "82",
                    "imdbRating": "9.0",
                    "imdbVotes": "1,652,832",
                    "imdbID": "tt0468569",
                    "Type": "movie",
                    "Response": "True"
                },
                {
                    "Title": "Batman Begins",
                    "Year": "2005",
                    "Rated": "PG-13",
                    "Released": "15 Jun 2005",
                    "Runtime": "140 min",
                    "Genre": "Action, Adventure",
                    "Director": "Christopher Nolan",
                    "Writer": "Bob Kane (characters), David S. Goyer (story), Christopher Nolan
(screenplay), David S. Goyer (screenplay)",
                    "Actors": "Christian Bale, Michael Caine, Liam Neeson, Katie Holmes",
                    "Plot": "After training with his mentor, Batman begins his fight to free crime-ridden
```

```
Gotham City from the corruption that Scarecrow and the League of Shadows have cast upon it.",
                    "Language": "English, Urdu, Mandarin",
                    "Country": "USA, UK",
                    "Awards": "Nominated for 1 Oscar. Another 15 wins & 66 nominations.",
                    "Poster": "http://ia.media-
imdb.com/images/M/MV5BNTM3OTc0MzM2OV5BMl5BanBnXkFtZTYwNzUwMTI3._V1_SX300.jpg",
                    "Metascore": "70",
                    "imdbRating": "8.3",
                    "imdbVotes": "972,584",
                    "imdbID": "tt0372784",
                    "Type": "movie",
                    "Response": "True"
                },
                {
                    "Title": "Avatar",
                    "Year": "2009",
                    "Rated": "PG-13",
                    "Released": "18 Dec 2009",
                    "Runtime": "162 min",
                    "Genre": "Action, Adventure, Fantasy",
                    "Director": "James Cameron",
                    "Writer": "James Cameron",
                    "Actors": "Sam Worthington, Zoe Saldana, Sigourney Weaver, Stephen Lang",
                    "Plot": "A paraplegic marine dispatched to the moon Pandora on a unique mission
becomes torn between following his orders and protecting the world he feels is his home.",
                    "Language": "English, Spanish",
                    "Country": "USA, UK",
                    "Awards": "Won 3 Oscars. Another 80 wins & 121 nominations.",
                    "Poster": "http://ia.media-
imdb.com/images/M/MV5BMTYwOTEwNjAzMl5BMl5BanBnXkFtZTcwODc5MTUwMw@@._V1_SX300.jpg",
                    "Metascore": "83",
                    "imdbRating": "7.9",
                    "imdbVotes": "876,575",
                    "imdbID": "tt0499549",
                    "Type": "movie",
                    "Response": "True"
                }
];

function getRating(watchList){
  var averageRating;
  const rating = watchList
                  .filter(obj => obj.Director === "Christopher Nolan")
                  .map(obj => Number(obj.imdbRating));
  averageRating = rating.reduce((accum, curr) => accum + curr)/rating.length;
  return averageRating;
}
```

16. Sort an Array Alphabetically using the sort Method

Description

The `sort` method sorts the elements of an array according to the callback function. For example:

```
function ascendingOrder(arr) {
  return arr.sort(function(a, b) {
    return a - b;
  });
}
ascendingOrder([1, 5, 2, 3, 4]);
// Returns [1, 2, 3, 4, 5]

function reverseAlpha(arr) {
  return arr.sort(function(a, b) {
    return a === b ? 0 : a < b ? 1 : -1;
  });
}
reverseAlpha(['l', 'h', 'z', 'b', 's']);
// Returns ['z', 's', 'l', 'h', 'b']
```

Note: It's encouraged to provide a callback function to specify how to sort the array items. JavaScript's default sorting method is by string Unicode point value, which may return unexpected results.

Instructions

Use the `sort` method in the `alphabeticalOrder` function to sort the elements of `arr` in alphabetical order.

Challenge Seed

```
function alphabeticalOrder(arr) {
  // Add your code below this line


  // Add your code above this line
}
alphabeticalOrder(["a", "d", "c", "a", "z", "g"]);
```

Solution

```
// solution required
```

17. Return a Sorted Array Without Changing the Original Array

Description

A side effect of the `sort` method is that it changes the order of the elements in the original array. In other words, it mutates the array in place. One way to avoid this is to first concatenate an empty array to the one being sorted (remember that `concat` returns a new array), then run the `sort` method.

Instructions

Use the `sort` method in the `nonMutatingSort` function to sort the elements of an array in ascending order. The function should return a new array, and not mutate the `globalArray` variable.

Challenge Seed

```
var globalArray = [5, 6, 3, 2, 9];
function nonMutatingSort(arr) {
  // Add your code below this line


  // Add your code above this line
}
nonMutatingSort(globalArray);
```

Solution

```
// solution required
```

18. Split a String into an Array Using the split Method

Description

The `split` method splits a string into an array of strings. It takes an argument for the delimiter, which can be a character to use to break up the string or a regular expression. For example, if the delimiter is a space, you get an array of words, and if the delimiter is an empty string, you get an array of each character in the string. Here are two examples that split one string by spaces, then another by digits using a regular expression:

```
var str = "Hello World";
var bySpace = str.split(" ");
// Sets bySpace to ["Hello", "World"]

var otherString = "How9are7you2today";
var byDigits = otherString.split(/\d/);
// Sets byDigits to ["How", "are", "you", "today"]
```

Since strings are immutable, the `split` method makes it easier to work with them.

Instructions

Use the `split` method inside the `splitify` function to split `str` into an array of words. The function should return the array. Note that the words are not always separated by spaces, and the array should not contain punctuation.

Challenge Seed

```
function splitify(str) {
  // Add your code below this line


  // Add your code above this line
}
splitify("Hello World,I-am code");
```

Solution

```
// solution required
```

19. Combine an Array into a String Using the join Method

Description

The `join` method is used to join the elements of an array together to create a string. It takes an argument for the delimiter that is used to separate the array elements in the string. Here's an example:

```
var arr = ["Hello", "World"];
var str = arr.join(" ");
// Sets str to "Hello World"
```

Instructions

Use the `join` method (among others) inside the `sentensify` function to make a sentence from the words in the string `str`. The function should return a string. For example, "I-like-Star-Wars" would be converted to "I like Star Wars". For this challenge, do not use the `replace` method.

Challenge Seed

```
function sentensify(str) {
  // Add your code below this line


  // Add your code above this line
}
sentensify("May-the-force-be-with-you");
```

Solution

```
// solution required
```

20. Apply Functional Programming to Convert Strings to URL Slugs

Description

The last several challenges covered a number of useful array and string methods that follow functional programming principles. We've also learned about `reduce`, which is a powerful method used to reduce problems to simpler forms. From computing averages to sorting, any array operation can be achieved by applying it. Recall that `map` and `filter` are special cases of `reduce`. Let's combine what we've learned to solve a practical problem. Many content management sites (CMS) have the titles of a post added to part of the URL for simple bookmarking purposes. For example, if you write a Medium post titled "Stop Using Reduce", it's likely the URL would have some form of the title string in it (".../stop-using-reduce"). You may have already noticed this on the freeCodeCamp site.

Instructions

Fill in the `urlSlug` function so it converts a string `title` and returns the hyphenated version for the URL. You can use any of the methods covered in this section, and don't use `replace`. Here are the requirements: The input is a string with spaces and title-cased words The output is a string with the spaces between words replaced by a hyphen (`-`) The output should be all lower-cased letters The output should not have any spaces

Challenge Seed

```
// the global variable
var globalTitle = "Winter Is Coming";

// Add your code below this line
function urlSlug(title) {


}
// Add your code above this line

var winterComing = urlSlug(globalTitle); // Should be "winter-is-coming"
```

Solution

```
// solution required
```

21. Use the every Method to Check that Every Element in an Array Meets a Criteria

Description

The `every` method works with arrays to check if *every* element passes a particular test. It returns a Boolean value - `true` if all values meet the criteria, `false` if not. For example, the following code would check if every element in the `numbers` array is less than 10:

```
var numbers = [1, 5, 8, 0, 10, 11];
numbers.every(function(currentValue) {
  return currentValue < 10;
});
// Returns false
```

Instructions

Use the `every` method inside the `checkPositive` function to check if every element in `arr` is positive. The function should return a Boolean value.

Challenge Seed

```
function checkPositive(arr) {
  // Add your code below this line


  // Add your code above this line
}
checkPositive([1, 2, 3, -4, 5]);
```

Solution

```
// solution required
```

22. Use the some Method to Check that Any Elements in an Array Meet a Criteria

Description

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 `numbers` array is less than 10:

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

Instructions

Use the `some` method inside the `checkPositive` function to check if any element in `arr` is positive. The function should return a Boolean value.

Challenge Seed

```
function checkPositive(arr) {
  // Add your code below this line


  // Add your code above this line
```

```
}
checkPositive([1, 2, 3, -4, 5]);
```

Solution

```
// solution required
```

23. Introduction to Currying and Partial Application

Description

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;
  }
}
//Alternative using ES6
const curried = x => y => 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 `curried` function 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
```

Instructions

Fill in the body of the `add` function so it uses currying to add parameters `x`, `y`, and `z`.

Challenge Seed

```
function add(x) {
  // Add your code below this line


  // Add your code above this line
```

```
}
add(10)(20)(30);
```

Solution

```
const add = x => y => z => x + y + z
```

Intermediate Algorithm Scripting

1. Sum All Numbers in a Range

Description

We'll pass you an array of two numbers. Return the sum of those two numbers plus the sum of all the numbers between them. The lowest number will not always come first. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function sumAll(arr) {
  return 1;
}

sumAll([1, 4]);
```

Solution

```
function sumAll(arr) {
  var sum = 0;
  arr.sort(function(a,b) {return a-b;});
  for (var i = arr[0]; i <= arr[1]; i++) {
    sum += i;
  }
  return sum;
}
```

2. Diff Two Arrays

Description

**Compare two arrays and return a new array with any items only found in one of the two given arrays, but not both. In other words, return the symmetric difference of the two arrays. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code. Note
You can return the array with its elements in any order.**

Instructions

Challenge Seed

```
function diffArray(arr1, arr2) {
  var newArr = [];
  // Same, same; but different.
  return newArr;
}

diffArray([1, 2, 3, 5], [1, 2, 3, 4, 5]);
```

Solution

```
function diffArray(arr1, arr2) {
  var newArr = [];
  var h1 = Object.create(null);
  arr1.forEach(function(e) {
    h1[e] = e;
  });

  var h2 = Object.create(null);
  arr2.forEach(function(e) {
    h2[e] = e;
  });

  Object.keys(h1).forEach(function(e) {
     if (!(e in h2)) newArr.push(h1[e]);
  });
  Object.keys(h2).forEach(function(e) {
     if (!(e in h1)) newArr.push(h2[e]);
  });
  // Same, same; but different.
  return newArr;
}
```

3. Seek and Destroy

Description

You will be provided with an initial array (the first argument in the destroyer function), followed by one or more arguments. Remove all elements from the initial array that are of the same value as these arguments. Note You have to use the `arguments` object. Remember to use Read-Search-Ask if you get stuck. Write your own code.

Instructions

Challenge Seed

```
function destroyer(arr) {
  // Remove all the values
  return arr;
}

destroyer([1, 2, 3, 1, 2, 3], 2, 3);
```

Solution

```
function destroyer(arr) {
  var hash = Object.create(null);
  [].slice.call(arguments, 1).forEach(function(e) {
    hash[e] = true;
  });
  // Remove all the values
  return arr.filter(function(e) { return !(e in hash);});
}
```

```
destroyer([1, 2, 3, 1, 2, 3], 2, 3);
```

4. Wherefore art thou

Description

Make a function that looks through an array of objects (first argument) and returns an array of all objects that have matching name and value pairs (second argument). Each name and value pair of the source object has to be present in the object from the collection if it is to be included in the returned array. For example, if the first argument is `[{ first: "Romeo", last: "Montague" }, { first: "Mercutio", last: null }, { first: "Tybalt", last: "Capulet" }]`, and the second argument is `{ last: "Capulet" }`, then you must return the third object from the array (the first argument), because it contains the name and its value, that was passed on as the second argument. Remember to use Read-Search-Ask if you get stuck. Write your own code.

Instructions

Challenge Seed

```
function whatIsInAName(collection, source) {
  // What's in a name?
  var arr = [];
  // Only change code below this line


  // Only change code above this line
  return arr;
}

whatIsInAName([{ first: "Romeo", last: "Montague" }, { first: "Mercutio", last: null }, { first:
"Tybalt", last: "Capulet" }], { last: "Capulet" });
```

Solution

```
function whatIsInAName(collection, source) {
  var arr = [];
  var keys = Object.keys(source);
  collection.forEach(function(e) {
    if(keys.every(function(key) {return e[key] === source[key];})) {
      arr.push(e);
    }
  });
  return arr;
}
```

5. Spinal Tap Case

Description

Convert a string to spinal case. Spinal case is all-lowercase-words-joined-by-dashes. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function spinalCase(str) {
  // "It's such a fine line between stupid, and clever."
  // --David St. Hubbins
  return str;
}

spinalCase('This Is Spinal Tap');
```

Solution

```
function spinalCase(str) {
  // "It's such a fine line between stupid, and clever."
  // --David St. Hubbins
  str = str.replace(/([a-z](?=[A-Z]))/g, '$1 ');
  return str.toLowerCase().replace(/\ |\_/g, '-');
}
```

6. Pig Latin

Description

Translate the provided string to pig latin. Pig Latin takes the first consonant (or consonant cluster) of an English word, moves it to the end of the word and suffixes an "ay". If a word begins with a vowel you just add "way" to the end. If a word does not contain a vowel, just add "ay" to the end. Input strings are guaranteed to be English words in all lowercase. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function translatePigLatin(str) {
  return str;
}

translatePigLatin("consonant");
```

Solution

```
function translatePigLatin(str) {
  if (isVowel(str.charAt(0))) return str + "way";
  var front = [];
  str = str.split('');
  while (str.length && !isVowel(str[0])) {
    front.push(str.shift());
  }
  return [].concat(str, front).join('') + 'ay';
}

function isVowel(c) {
  return ['a', 'e', 'i', 'o', 'u'].indexOf(c.toLowerCase()) !== -1;
}
```

7. Search and Replace

Description

Perform a search and replace on the sentence using the arguments provided and return the new sentence. First argument is the sentence to perform the search and replace on. Second argument is the word that you will be replacing (before). Third argument is what you will be replacing the second argument with (after). Note Preserve the case of the first character in the original word when you are replacing it. For example if you mean to replace the word "Book" with the word "dog", it should be replaced as "Dog" Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function myReplace(str, before, after) {
  return str;
}

myReplace("A quick brown fox jumped over the lazy dog", "jumped", "leaped");
```

Solution

```
function myReplace(str, before, after) {
  if (before.charAt(0) === before.charAt(0).toUpperCase()) {
    after = after.charAt(0).toUpperCase() + after.substring(1);
  } else {
    after = after.charAt(0).toLowerCase() + after.substring(1);
  }
  return str.replace(before, after);
}
```

8. DNA Pairing

Description

The DNA strand is missing the pairing element. Take each character, get its pair, and return the results as a 2d array. Base pairs are a pair of AT and CG. Match the missing element to the provided character. Return the provided character as the first element in each array. For example, for the input GCG, return [["G", "C"], ["C","G"], ["G", "C"]] The character and its pair are paired up in an array, and all the arrays are grouped into one encapsulating array. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function pairElement(str) {
  return str;
}

pairElement("GCG");
```

Solution

```
var lookup = Object.create(null);
lookup.A = 'T';
lookup.T = 'A';
lookup.C = 'G';
lookup.G = 'C';
```

```
function pairElement(str) {
 return str.split('').map(function(p) {return [p, lookup[p]];});
}
```

9. Missing letters

Description

Find the missing letter in the passed letter range and return it. If all letters are present in the range, return undefined. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function fearNotLetter(str) {
  return str;
}

fearNotLetter("abce");
```

Solution

```
function fearNotLetter (str) {
  for (var i = str.charCodeAt(0); i <= str.charCodeAt(str.length - 1); i++) {
    var letter = String.fromCharCode(i);
    if (str.indexOf(letter) === -1) {
      return letter;
    }
  }

  return undefined;
}
```

10. Sorted Union

Description

Write a function that takes two or more arrays and returns a new array of unique values in the order of the original provided arrays. In other words, all values present from all arrays should be included in their original order, but with no duplicates in the final array. The unique numbers should be sorted by their original order, but the final array should not be sorted in numerical order. Check the assertion tests for examples. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function uniteUnique(arr) {
  return arr;
}

uniteUnique([1, 3, 2], [5, 2, 1, 4], [2, 1]);
```

Solution

```
function uniteUnique(arr) {
  return [].slice.call(arguments).reduce(function(a, b) {
    return [].concat(a, b.filter(function(e) {return a.indexOf(e) === -1;}));
  }, []);
}
```

11. Convert HTML Entities

Description

Convert the characters `&`, `<`, `>`, `"` (double quote), and `'` (apostrophe), in a string to their corresponding HTML entities. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function convertHTML(str) {
  // &colon;&rpar;
  return str;
}

convertHTML("Dolce & Gabbana");
```

Solution

```
var MAP = { '&': '&amp;',
            '<': '&lt;',
            '>': '&gt;',
            '"': '&quot;',
            "'": '&apos;'};

function convertHTML(str) {
  return str.replace(/[&<>"']/g, function(c) {
    return MAP[c];
  });
}
```

12. Sum All Odd Fibonacci Numbers

Description

Given a positive integer `num`, return the sum of all odd Fibonacci numbers that are less than or equal to `num`. The first two numbers in the Fibonacci sequence are 1 and 1. Every additional number in the sequence is the sum of the two previous numbers. The first six numbers of the Fibonacci sequence are 1, 1, 2, 3, 5 and 8. For example, `sumFibs(10)` should return `10` because all odd Fibonacci numbers less than or equal to `10` are 1, 1, 3, and 5. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function sumFibs(num) {
  return num;
}

sumFibs(4);
```

Solution

```
function sumFibs(num) {
  var a = 1;
  var b = 1;
  var s = 0;
  while (a <= num) {
    if (a % 2 !== 0) {
      s += a;
    }
    a = [b, b=b+a][0];
  }
  return s;
}
```

13. Sum All Primes

Description

Sum all the prime numbers up to and including the provided number. A prime number is defined as a number greater than one and having only two divisors, one and itself. For example, 2 is a prime number because it's only divisible by one and two. The provided number may not be a prime. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function sumPrimes(num) {
  return num;
}

sumPrimes(10);
```

Solution

```
function eratosthenesArray(n) {
    var primes = [];
    if (n > 2) {
        var half = n>>1;
        var sieve = Array(half);
        for (var i = 1, limit = Math.sqrt(n)>>1; i <= limit; i++) {
            if (!sieve[i]) {
                for (var step = 2*i+1, j = (step*step)>>1; j < half; j+=step) {
                    sieve[j] = true;
                }
            }
        }
        primes.push(2);
        for (var p = 1; p < half; p++) {
            if (!sieve[p]) primes.push(2*p+1);
        }
    }
    return primes;
}
```

```
function sumPrimes(num) {
  return eratosthenesArray(num+1).reduce(function(a,b) {return a+b;}, 0);
}

sumPrimes(10);
```

14. Smallest Common Multiple

Description

Find the smallest common multiple of the provided parameters that can be evenly divided by both, as well as by all sequential numbers in the range between these parameters. The range will be an array of two numbers that will not necessarily be in numerical order. For example, if given 1 and 3, find the smallest common multiple of both 1 and 3 that is also evenly divisible by all numbers *between* 1 and 3. The answer here would be 6. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function smallestCommons(arr) {
  return arr;
}


smallestCommons([1,5]);
```

Solution

```
function gcd(a, b) {
    while (b !== 0) {
        a = [b, b = a % b][0];
    }
    return a;
}

function lcm(a, b) {
    return (a * b) / gcd(a, b);
}

function smallestCommons(arr) {
  arr.sort(function(a,b) {return a-b;});
  var rng = [];
  for (var i = arr[0]; i <= arr[1]; i++) {
    rng.push(i);
  }
  return rng.reduce(lcm);
}
```

15. Drop it

Description

Given the array `arr`, iterate through and remove each element starting from the first element (the 0 index) until the function `func` returns `true` when the iterated element is passed through it. Then return the rest of the array once the condition is satisfied, otherwise, `arr` should be returned as an empty array. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function dropElements(arr, func) {
  // Drop them elements.
  return arr;
}

dropElements([1, 2, 3], function(n) {return n < 3; });
```

Solution

```
function dropElements(arr, func) {
  // Drop them elements.
  while (arr.length && !func(arr[0])) {
    arr.shift();
  }
  return arr;
}
```

16. Steamroller

Description

Flatten a nested array. You must account for varying levels of nesting. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function steamrollArray(arr) {
  // I'm a steamroller, baby
  return arr;
}

steamrollArray([1, [2], [3, [[4]]]]);
```

Solution

```
function steamrollArray(arr) {
  if (!Array.isArray(arr)) {
    return [arr];
  }
  var out = [];
  arr.forEach(function(e) {
    steamrollArray(e).forEach(function(v) {
      out.push(v);
    });
  });
  return out;
}
```

17. Binary Agents

Description

Return an English translated sentence of the passed binary string. The binary string will be space separated. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function binaryAgent(str) {
  return str;
}

binaryAgent("01000001 01110010 01100101 01101110 00100111 01110100 00100000 01100010 01101111 01101110
01100110 01101001 01110010 01100101 01110011 00100000 01100110 01110101 01101110 00100001 00111111");
```

Solution

```
function binaryAgent(str) {
  return str.split(' ').map(function(s) { return parseInt(s, 2); }).map(function(b) { return
String.fromCharCode(b);}).join('');
}
```

18. Everything Be True

Description

Check if the predicate (second argument) is truthy on all elements of a collection (first argument). In other words, you are given an array collection of objects. The predicate `pre` will be an object property and you need to return `true` if its value is `truthy`. Otherwise, return `false`. In JavaScript, `truthy` values are values that translate to `true` when evaluated in a Boolean context. Remember, you can access object properties through either dot notation or `[]` notation. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function truthCheck(collection, pre) {
  // Is everyone being true?
  return pre;
}

truthCheck([{"user": "Tinky-Winky", "sex": "male"}, {"user": "Dipsy", "sex": "male"}, {"user": "Laa-
Laa", "sex": "female"}, {"user": "Po", "sex": "female"}], "sex");
```

Solution

```
function truthCheck(collection, pre) {
  // Does everyone have one of these?
  return collection.every(function(e) { return e[pre]; });
}
```

19. Arguments Optional

Description

Create a function that sums two arguments together. If only one argument is provided, then return a function that expects one argument and returns the sum. For example, `addTogether(2, 3)` should return `5`, and `addTogether(2)` should return a function. Calling this returned function with a single argument will then return the sum: `var sumTwoAnd = addTogether(2);` `sumTwoAnd(3)` returns `5`. If either argument isn't a valid number, return undefined. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function addTogether() {
  return false;
}

addTogether(2,3);
```

Solution

```
function addTogether() {
  var a = arguments[0];
  if (toString.call(a) !== '[object Number]') return;
  if (arguments.length === 1) {
    return function(b) {
      if (toString.call(b) !== '[object Number]') return;
      return a + b;
    };
  }
  var b = arguments[1];
  if (toString.call(b) !== '[object Number]') return;
  return a + arguments[1];
}
```

20. Make a Person

Description

Fill in the object constructor with the following methods below:

> **getFirstName() getLastName() getFullName() setFirstName(first) setLastName(last) setFullName(firstAndLast)**

Run the tests to see the expected output for each method. The methods that take an argument must accept only one argument and it has to be a string. These methods must be the only available means of interacting with the object. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
var Person = function(firstAndLast) {
  // Complete the method below and implement the others similarly
  this.getFullName = function() {
    return "";
  };
```

```
  return firstAndLast;
};

var bob = new Person('Bob Ross');
bob.getFullName();
```

Solution

```
var Person = function(firstAndLast) {

  var firstName, lastName;

  function updateName(str) {
    firstName = str.split(" ")[0];
    lastName = str.split(" ")[1];
  }

  updateName(firstAndLast);

  this.getFirstName = function(){
    return firstName;
  };

  this.getLastName = function(){
    return lastName;
  };

  this.getFullName = function(){
    return firstName + " " + lastName;
  };

  this.setFirstName = function(str){
    firstName = str;
  };


  this.setLastName = function(str){
    lastName = str;
  };

  this.setFullName = function(str){
    updateName(str);
  };
};

var bob = new Person('Bob Ross');
bob.getFullName();
```

21. Map the Debris

Description

Return a new array that transforms the elements' average altitude into their orbital periods (in seconds). The array will contain objects in the format `{name: 'name', avgAlt: avgAlt}`. You can read about orbital periods on Wikipedia. The values should be rounded to the nearest whole number. The body being orbited is Earth. The radius of the earth is 6367.4447 kilometers, and the GM value of earth is 398600.4418 km^3s^{-2}. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function orbitalPeriod(arr) {
  var GM = 398600.4418;
```

```
  var earthRadius = 6367.4447;
  return arr;
}

orbitalPeriod([{name : "sputnik", avgAlt : 35873.5553}]);
```

Solution

```
function orbitalPeriod(arr) {
  var GM = 398600.4418;
  var earthRadius = 6367.4447;
  var TAU = 2 * Math.PI;
  return arr.map(function(obj) {
    return {
      name: obj.name,
      orbitalPeriod: Math.round(TAU * Math.sqrt(Math.pow(obj.avgAlt+earthRadius, 3)/GM))
    };
  });
}

orbitalPeriod([{name : "sputkin", avgAlt : 35873.5553}]);
```

JavaScript Algorithms and Data Structures Projects

1. Palindrome Checker

Description

Return `true` if the given string is a palindrome. Otherwise, return `false`. A palindrome is a word or sentence that's spelled the same way both forward and backward, ignoring punctuation, case, and spacing. Note You'll need to remove all non-alphanumeric characters (punctuation, spaces and symbols) and turn everything into the same case (lower or upper case) in order to check for palindromes. We'll pass strings with varying formats, such as `"racecar"`, `"RaceCar"`, and `"race CAR"` among others. We'll also pass strings with special symbols, such as `"2A3*3a2"`, `"2A3 3a2"`, and `"2_A3*3#A2"`. Remember to use Read-Search-Ask if you get stuck. Write your own code.

Instructions

Challenge Seed

```
function palindrome(str) {
  // Good luck!
  return true;
}



palindrome("eye");
```

Solution

```
function palindrome(str) {
  var string = str.toLowerCase().split(/[^A-Za-z0-9]/gi).join('');
  var aux = string.split('');
  if (aux.join('') === aux.reverse().join('')){
    return true;
  }
```

```
  return false;
}
```

2. Roman Numeral Converter

Description

Convert the given number into a roman numeral. All roman numerals answers should be provided in upper-case. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function convertToRoman(num) {
 return num;
}

convertToRoman(36);
```

Solution

```
function convertToRoman(num) {
  var ref = [['M', 1000], ['CM', 900], ['D', 500], ['CD', 400], ['C', 100], ['XC', 90], ['L', 50],
['XL', 40], ['X', 10], ['IX', 9], ['V', 5], ['IV', 4], ['I', 1]];
  var res = [];
  ref.forEach(function(p) {
    while (num >= p[1]) {
      res.push(p[0]);
      num -= p[1];
    }
  });
  return res.join('');
}
```

3. Caesars Cipher

Description

One of the simplest and most widely known ciphers is a `Caesar cipher` , also known as a `shift cipher` . In a `shift cipher` the meanings of the letters are shifted by some set amount. A common modern use is the ROT13 cipher, where the values of the letters are shifted by 13 places. Thus 'A' ↔ 'N', 'B' ↔ 'O' and so on. Write a function which takes a ROT13 encoded string as input and returns a decoded string. All letters will be uppercase. Do not transform any non-alphabetic character (i.e. spaces, punctuation), but do pass them on. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function rot13(str) { // LBH QVQ VG!

  return str;
}
```

```
// Change the inputs below to test
rot13("SERR PBQR PNZC");
```

Solution

```
var lookup = {
  'A': 'N','B': 'O','C': 'P','D': 'Q',
  'E': 'R','F': 'S','G': 'T','H': 'U',
  'I': 'V','J': 'W','K': 'X','L': 'Y',
  'M': 'Z','N': 'A','O': 'B','P': 'C',
  'Q': 'D','R': 'E','S': 'F','T': 'G',
  'U': 'H','V': 'I','W': 'J','X': 'K',
  'Y': 'L','Z': 'M'
};

function rot13(encodedStr) {
  var codeArr = encodedStr.split("");  // String to Array
  var decodedArr = []; // Your Result goes here
  // Only change code below this line

  decodedArr = codeArr.map(function(letter) {
    if(lookup.hasOwnProperty(letter)) {
      letter = lookup[letter];
    }
    return letter;
  });

  // Only change code above this line
  return decodedArr.join(""); // Array to String
}
```

4. Telephone Number Validator

Description

Return `true` if the passed string looks like a valid US phone number. The user may fill out the form field any way they choose as long as it has the format of a valid US number. The following are examples of valid formats for US numbers (refer to the tests below for other variants):

> **555-555-5555**
> **(555)555-5555**
> **(555) 555-5555**
> **555 555 5555**
> **5555555555**
> **1 555 555 5555**

For this challenge you will be presented with a string such as `800-692-7753` or `8oo-six427676;laskdjf`. Your job is to validate or reject the US phone number based on any combination of the formats provided above. The area code is required. If the country code is provided, you must confirm that the country code is `1`. Return `true` if the string is a valid US phone number; otherwise return `false`. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Instructions

Challenge Seed

```
function telephoneCheck(str) {
  // Good luck!
  return true;
}

telephoneCheck("555-555-5555");
```

Solution

```
var re = /^([+]?1[\s]?)?((?:[(](?:[2-9]1[02-9]|[2-9][02-8][0-9])[)][\s]?)|(?:(?:[2-9]1[02-9]|[2-9][02-8][0-9])[\s.-]?)){1}([2-9]1[02-9]|[2-9][02-9]1|[2-9][02-9]{2}[\s.-]?){1}([0-9]{4}){1}$/;

function telephoneCheck(str) {
  return re.test(str);
}

telephoneCheck("555-555-5555");
```

5. Cash Register

Description

Design a cash register drawer function `checkCashRegister()` that accepts purchase price as the first argument (`price`), payment as the second argument (`cash`), and cash-in-drawer (`cid`) as the third argument. `cid` is a 2D array listing available currency. The `checkCashRegister()` function should always return an object with a `status` key and a `change` key. Return `{status: "INSUFFICIENT_FUNDS", change: []}` if cash-in-drawer is less than the change due, or if you cannot return the exact change. Return `{status: "CLOSED", change: [...]}` with cash-in-drawer as the value for the key `change` if it is equal to the change due. Otherwise, return `{status: "OPEN", change: [...]}`, with the change due in coins and bills, sorted in highest to lowest order, as the value of the `change` key. Remember to use Read-Search-Ask if you get stuck. Try to pair program. Write your own code.

Currency Unit	Amount
Penny	$0.01 (PENNY)
Nickel	$0.05 (NICKEL)
Dime	$0.1 (DIME)
Quarter	$0.25 (QUARTER)
Dollar	$1 (DOLLAR)
Five Dollars	$5 (FIVE)
Ten Dollars	$10 (TEN)
Twenty Dollars	$20 (TWENTY)
One-hundred Dollars	$100 (ONE HUNDRED)

Instructions

Challenge Seed

```
function checkCashRegister(price, cash, cid) {
  var change;
  // Here is your change, ma'am.
  return change;
}

// Example cash-in-drawer array:
// [["PENNY", 1.01],
// ["NICKEL", 2.05],
// ["DIME", 3.1],
// ["QUARTER", 4.25],
// ["ONE", 90],
// ["FIVE", 55],
// ["TEN", 20],
// ["TWENTY", 60],
// ["ONE HUNDRED", 100]]
```

```
checkCashRegister(19.5, 20, [["PENNY", 1.01], ["NICKEL", 2.05], ["DIME", 3.1], ["QUARTER", 4.25],
["ONE", 90], ["FIVE", 55], ["TEN", 20], ["TWENTY", 60], ["ONE HUNDRED", 100]]);
```

Solution

```
var denom = [
        { name: 'ONE HUNDRED', val: 100},
        { name: 'TWENTY', val: 20},
        { name: 'TEN', val: 10},
        { name: 'FIVE', val: 5},
        { name: 'ONE', val: 1},
        { name: 'QUARTER', val: 0.25},
        { name: 'DIME', val: 0.1},
        { name: 'NICKEL', val: 0.05},
        { name: 'PENNY', val: 0.01}
];

function checkCashRegister(price, cash, cid) {
 var output = {status: null, change: []};
 var change = cash - price;
 var register = cid.reduce(function(acc, curr) {
 acc.total += curr[1];
 acc[curr[0]] = curr[1];
 return acc;
 }, {total: 0});
 if(register.total === change) {
 output.status = 'CLOSED';
 output.change = cid;
 return output;
 }
 if(register.total < change) {
 output.status = 'INSUFFICIENT_FUNDS';
 return output;
 }
 var change_arr = denom.reduce(function(acc, curr) {
 var value = 0;
 while(register[curr.name] > 0 && change >= curr.val) {
 change -= curr.val;
 register[curr.name] -= curr.val;
 value += curr.val;
 change = Math.round(change * 100) / 100;
 }
 if(value > 0) {
 acc.push([ curr.name, value ]);
 }
 return acc;
 }, []);
 if(change_arr.length < 1 || change > 0) {
 output.status = 'INSUFFICIENT_FUNDS';
 return output;
 }
 output.status = 'OPEN';
 output.change = change_arr;
 return output;
}
```