JavaScript was originally designed in 1995 as a scripting language to make websites interactive. Over the last 25 years it has evolved to be a core building block of the modern web and beyond. JavaScript is now used in everything from web development, games, mobile apps, desktop software, server-side functionality and even space exploration technology. In 2020, GitHub (used by more than 50 million developers to build and maintain software) noted that JavaScript remains the most popular programming language in the world.

Why Learn to Code in JavaScript?

Beyond being ranked as the most popular coding language by GitHub, it's important to note that 70% of companies hiring developers are looking for people with JavaScript skills with average salaries above $100,000 a year.


JavaScript has become a powerful and versatile language used by the world's top companies, on their flagship products and services, including Facebook, Google, Microsoft, eBay, Netflix, Uber, NASA and SpaceX.

In this lesson we look at how to run JavaScript code. For that we require a **web browser** and **code editor**. When it comes to code editors there are many good choices for JavaScript. In this course we will be using **Visual Studio Code (VS Code)** which is free, widely used and available for all operating systems. We will also demonstrate **repl.it (**https://repl.it/**)** one of the many online options available for anyone wanting to get started without installing any software on your local computer. Follow along with the relevant VS Code installation (Windows or Mac) or skip past the VS Code sections to run your first line of JavaScript in an online environment.

VS Code Installation (Windows)

To install VS Code on Windows, browse to the VS Code homepage (https://code.visualstudio.com/) and follow the installation instructions for your operating system For Windows, click **Download for Windows** which will download the executable file to your computer. Once that completes, run the the installation **exe** and follow the installation prompts accepting or adjusting the default settings. When installation completes, you can run VS Code from the installation window or from the **Windows Start Menu**.
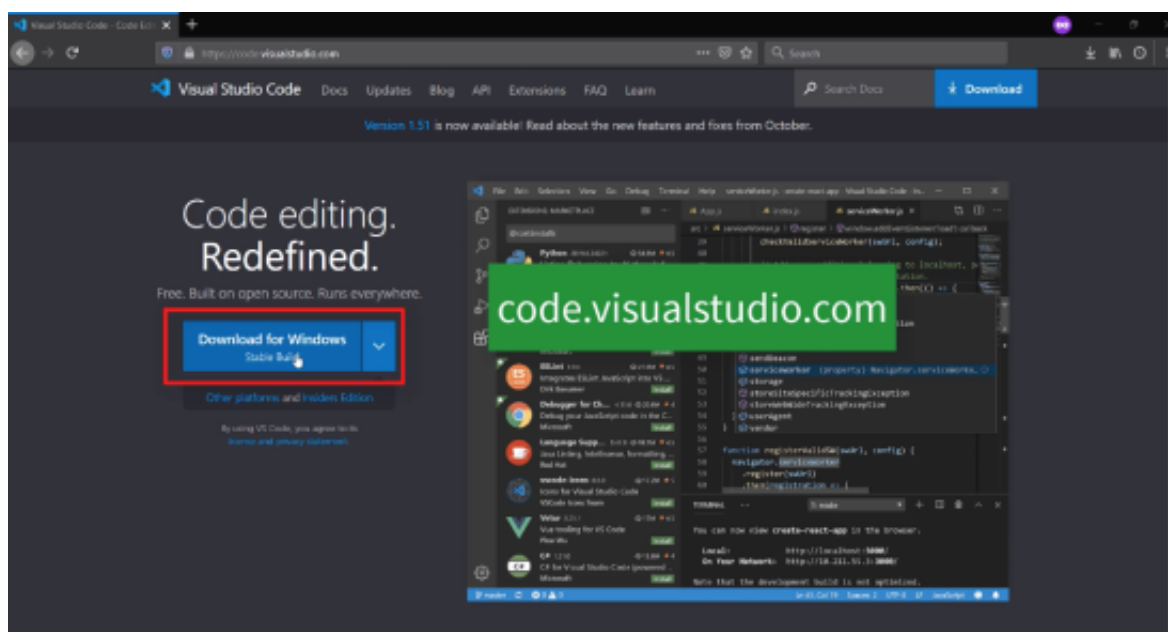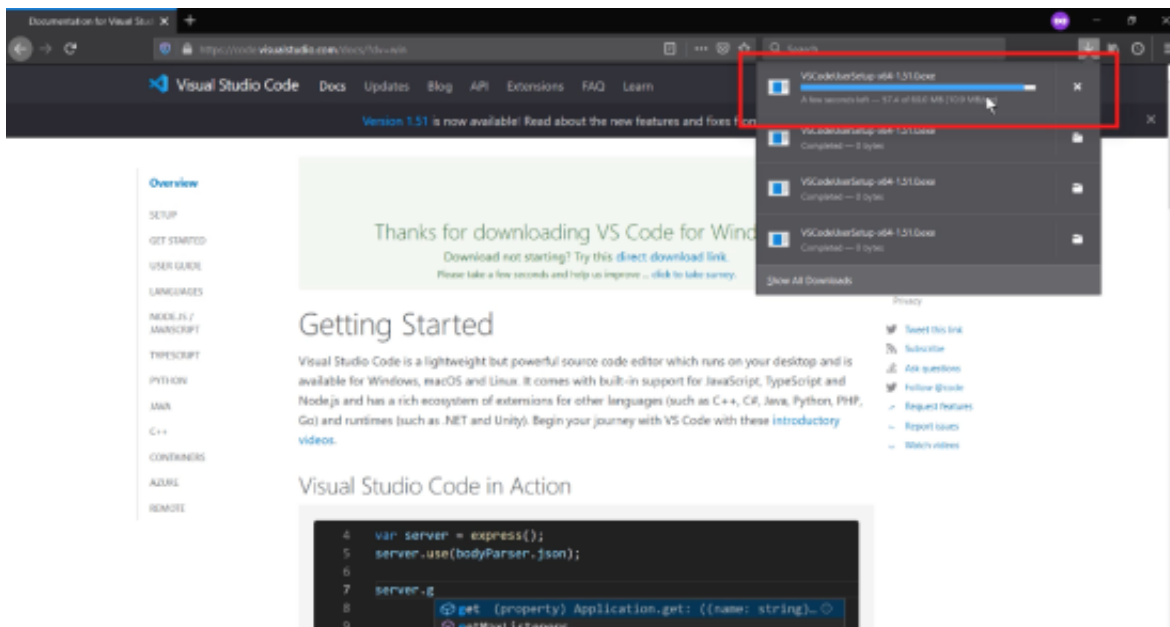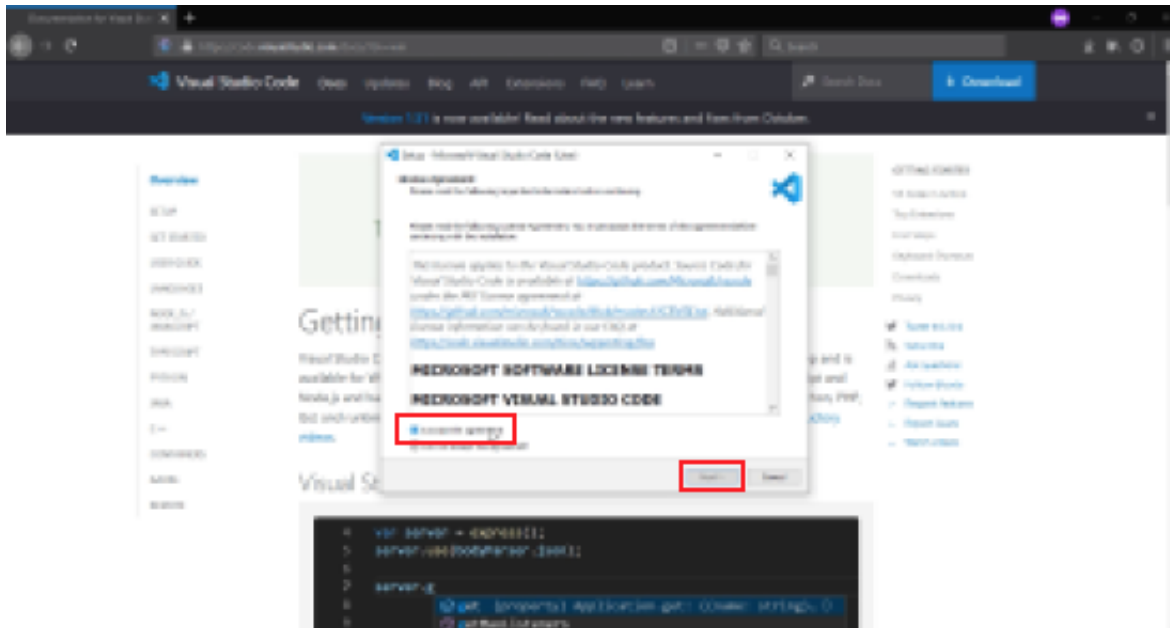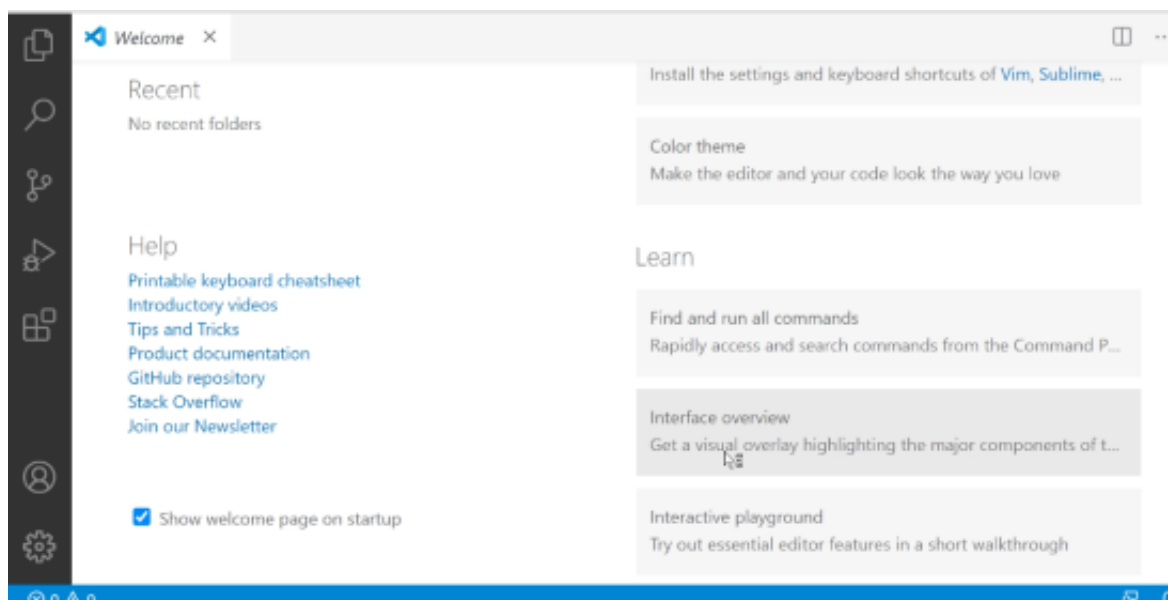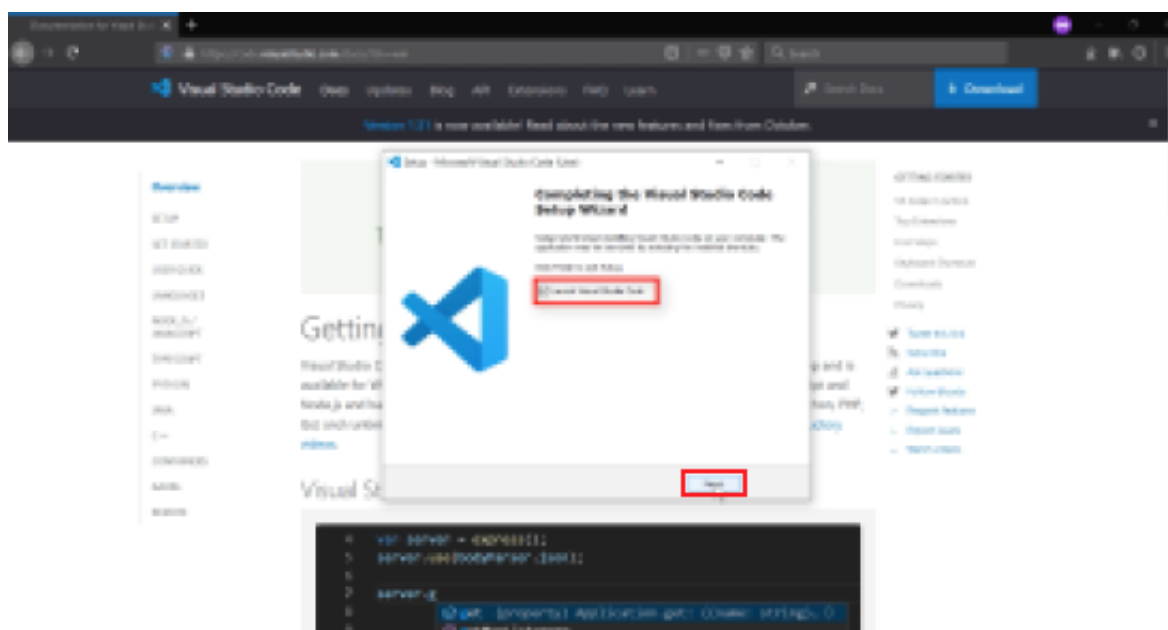
VS Code Installation (Mac)

To install VS Code on Mac, browse to the VS Code homepage (https://code.visualstudio.com/) and follow the installation instructions for your operating system. For Mac, click **Download for Mac** and select the option to **Open with Archive Utility** to extract it. When the file finishes downloading you should see the extracted **VS Code** file in your **Downloads** folder. From there, drag the VS Code file to the **Applications** folder if you want it in your **task bar** otherwise just **double-click** to open code editor. You may be prompted with a security warning. If so, click **open**.
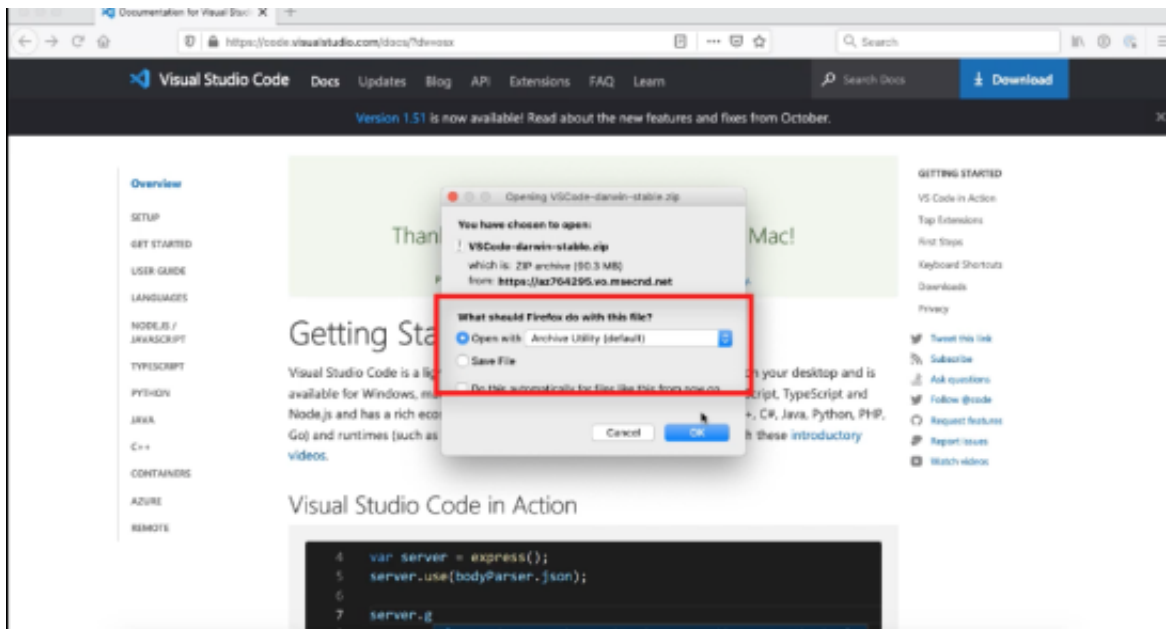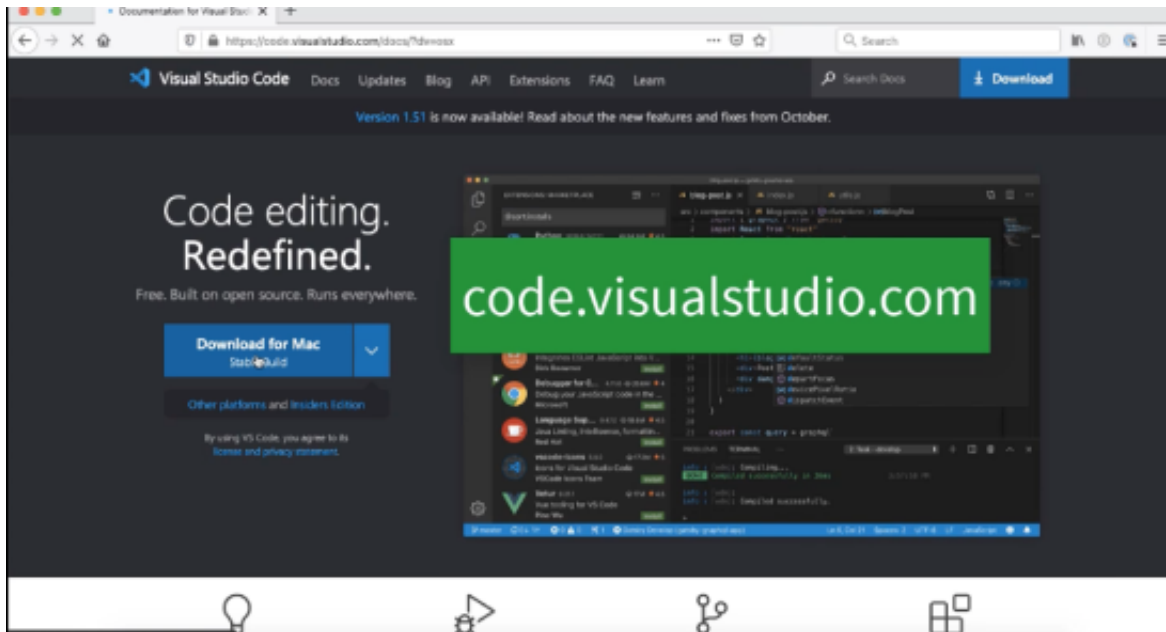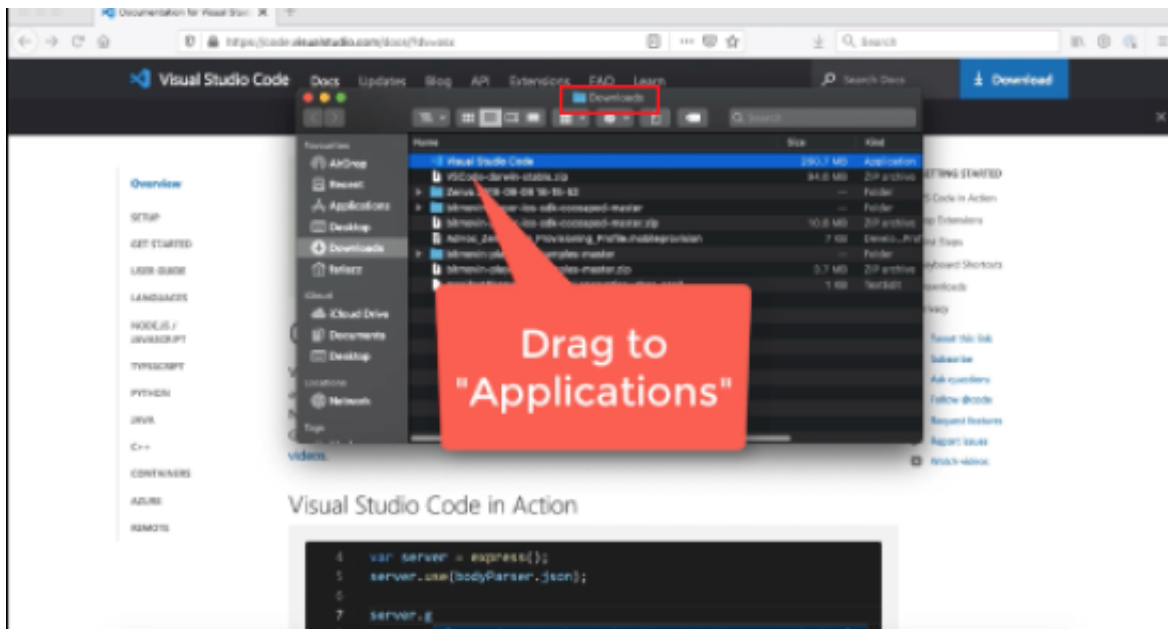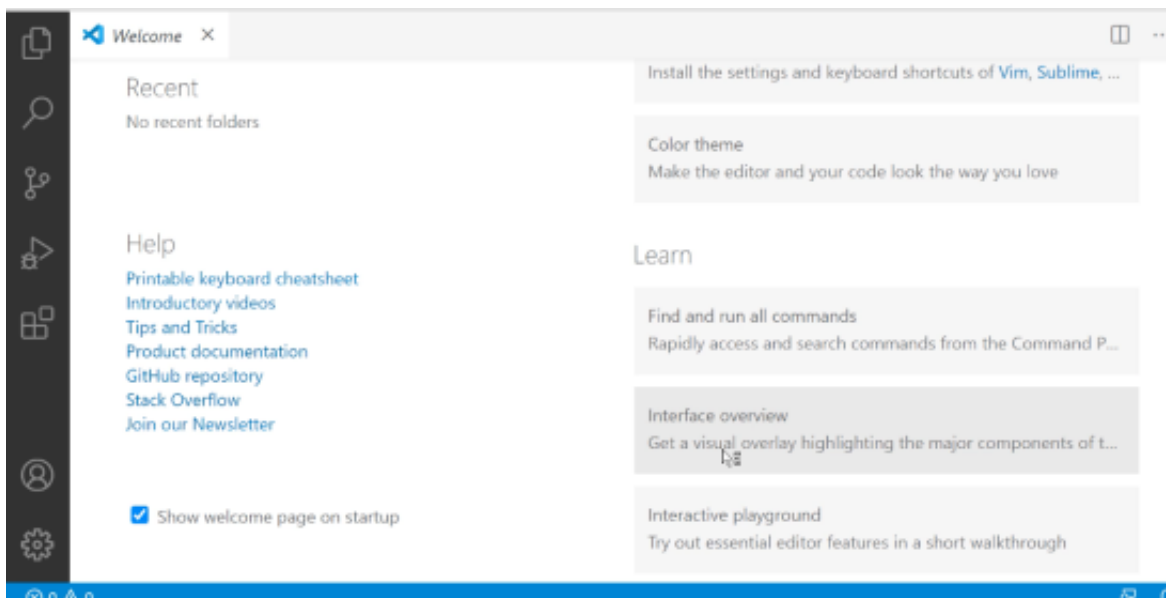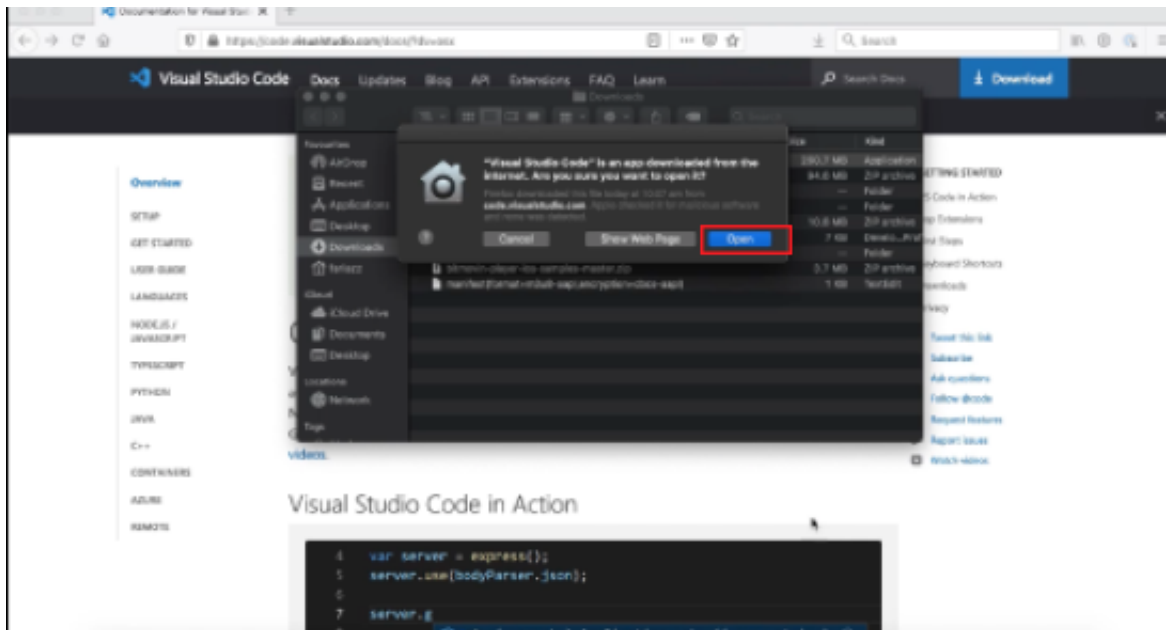
Demonstrate Simple JavaScript in VS Code

With your editor setup, choose a place to store your code files. If you don't already have a place, you can simply create a folder in your **Documents** or on your **Desktop** to get started. Inside VS Code, choose the **File** menu option and **Open Folder** menu item. In the example below, we create a new folder named **JS Course** inside our **Documents** folder (you are free to choose any location).
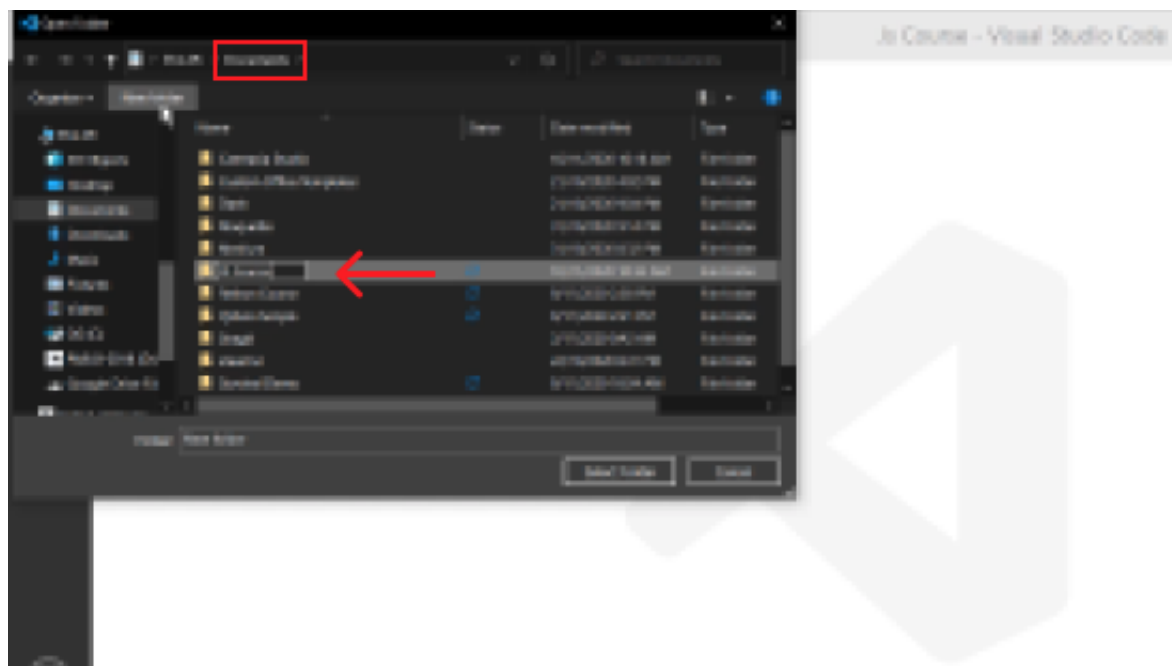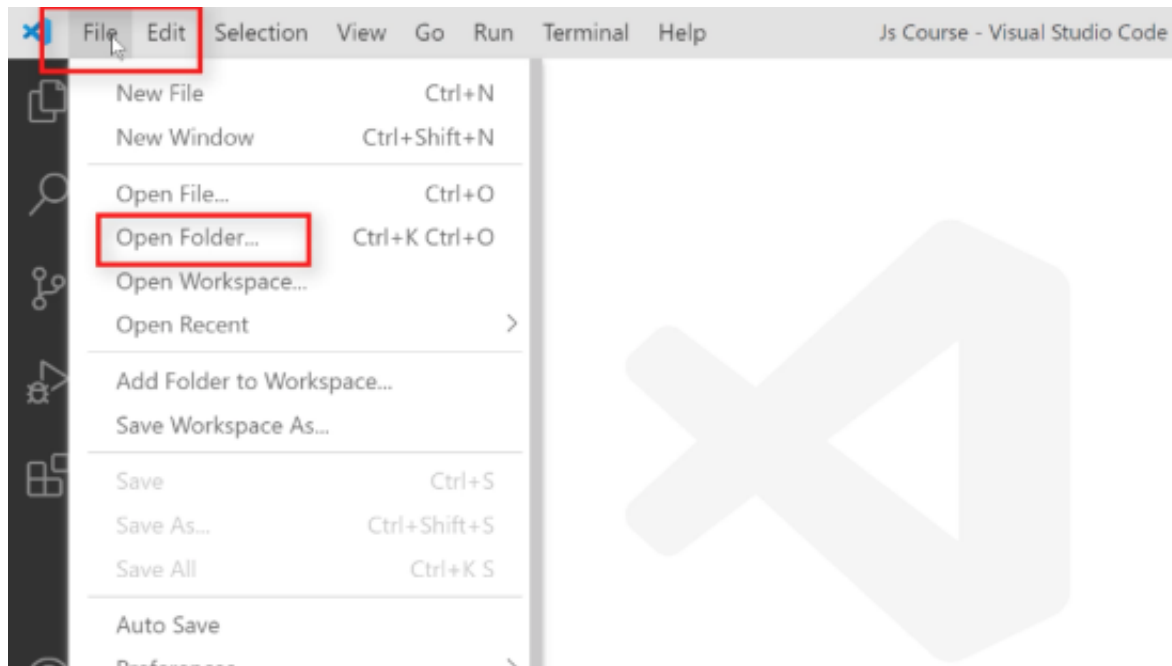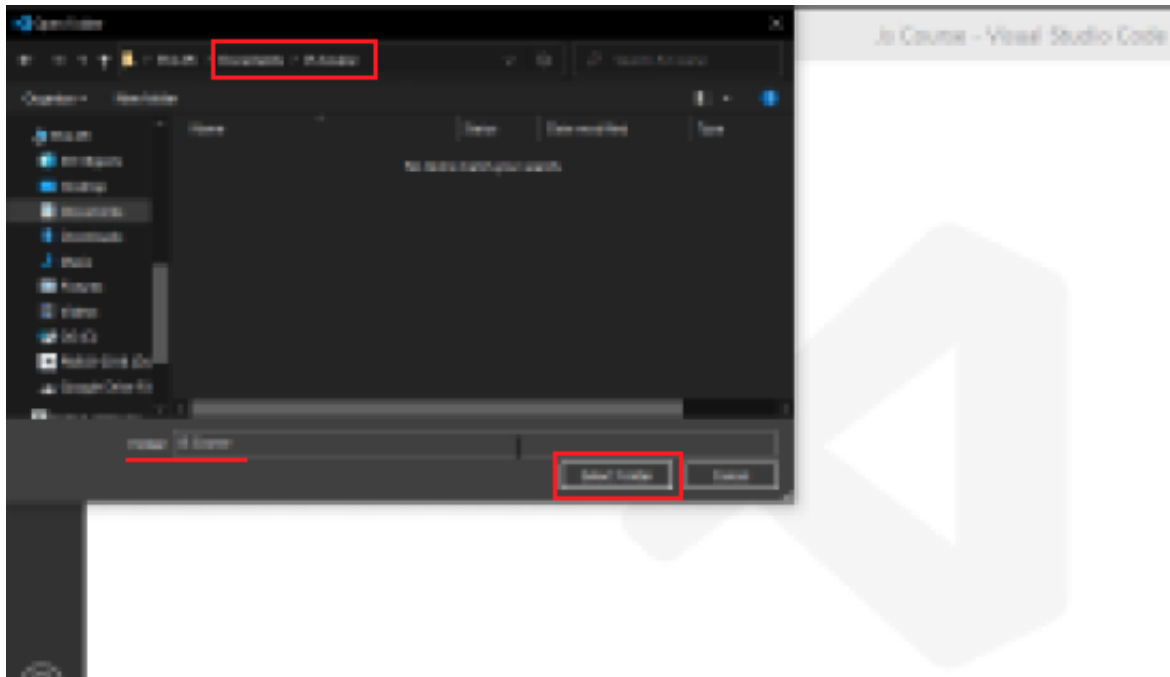
On the VS Code side bar you will see the **Explorer** on top. Click this icon to see the folder we just created. Once we create files for this project they will show here.
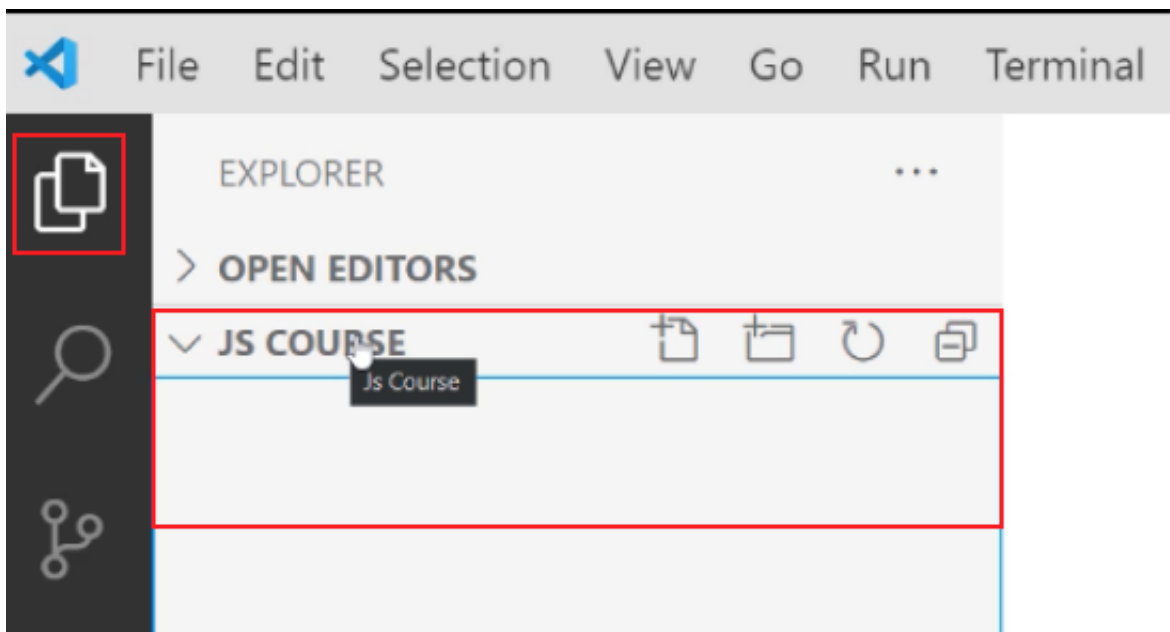
Create the first file by clicking on the **New File** icon (as shown below) or through the menu option choosing **File | New File**. Name the new file **index.html**.
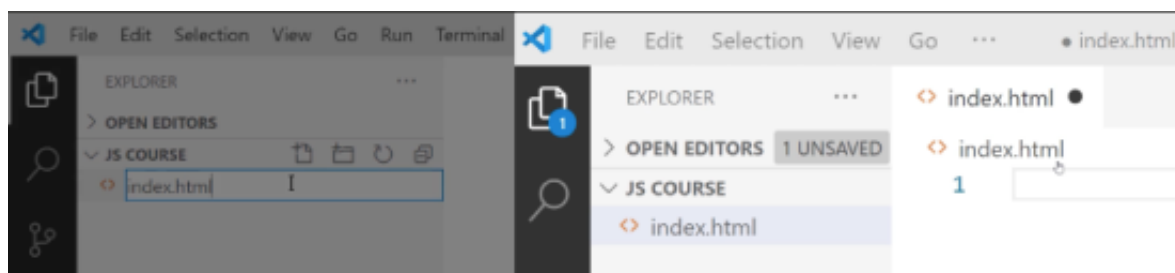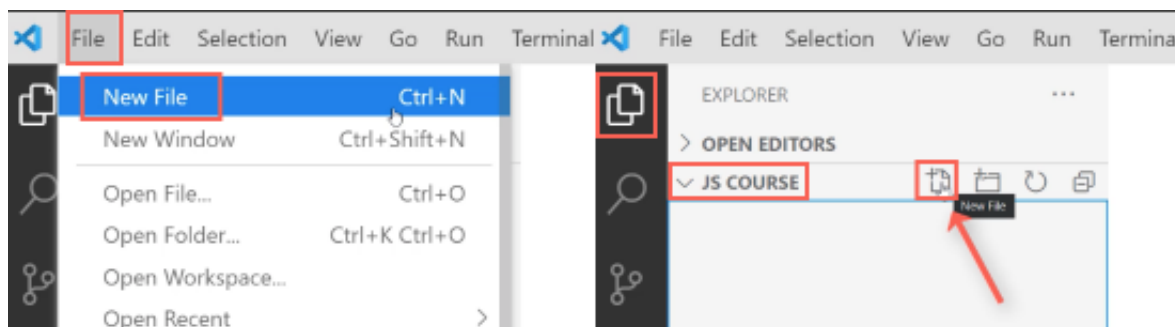
This **html** file represents a web page so anything written here will be displayed to the user. Type **hello world** into the **index.html** file.

```
hello world
```

There are many ways to open this file in a web browser. For simplicity either locate the file using **File Explorer** or right-click on the file in VS Code and choose **open in browser** from the context menu. The text we typed into the html page will appear in the browser.
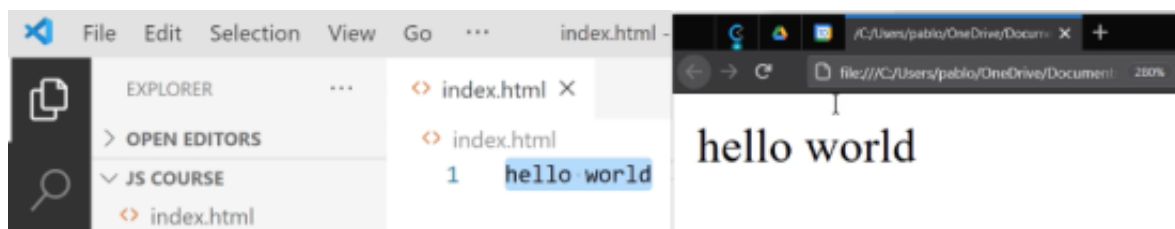
In order to run JavaScript from within a **html** file we must put inside **<script></script> tags**. Inside these script tags we can write JavaScript code such as the following command to display a **dialog box** to user with the text **Hello World**. Save the file (**control+s** on windows or **cmd+s** on mac) then open the file again in the web browser to see the text displayed to the user in a dialog box. We are now running JavaScript code in the browser.

```
<script>
    alert('Hello World')
</script>
```

Demonstrate Simple JavaScript in repl.it

As mentioned in the introduction, there is an alternative to installing software on your computer in order to write JavaScript code. We can use an online environment such as **repl.it**. If you want to try this option to go **(**https://repl.it/**)** and sign-up for a free account. Then, from the dashboard, create a **New REPL** with **HTML, CSS, JS** as the **Language** option.
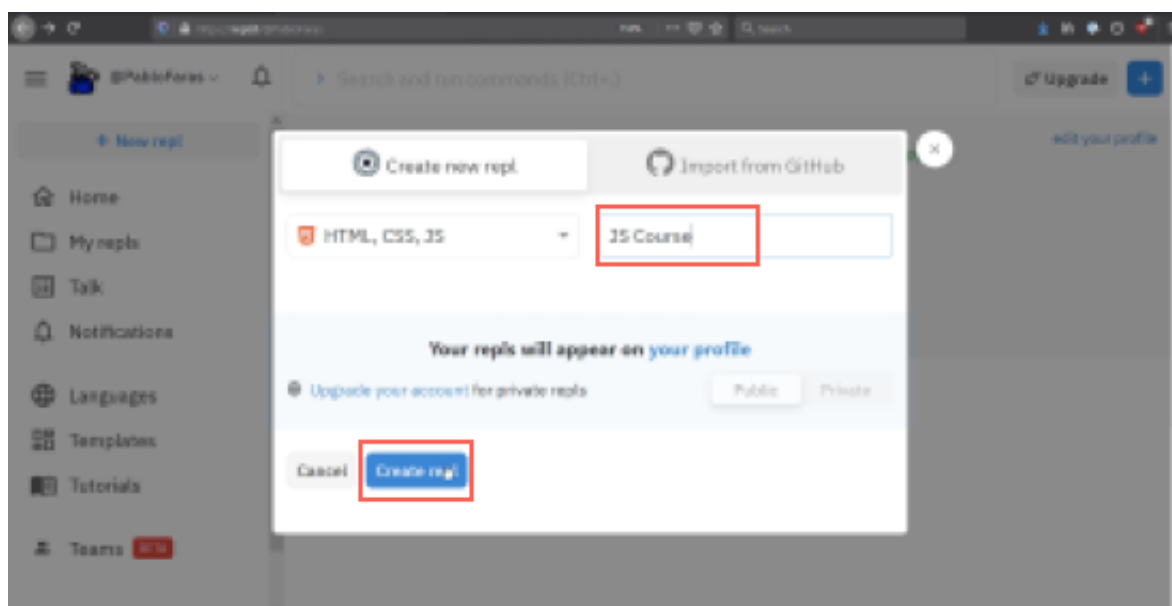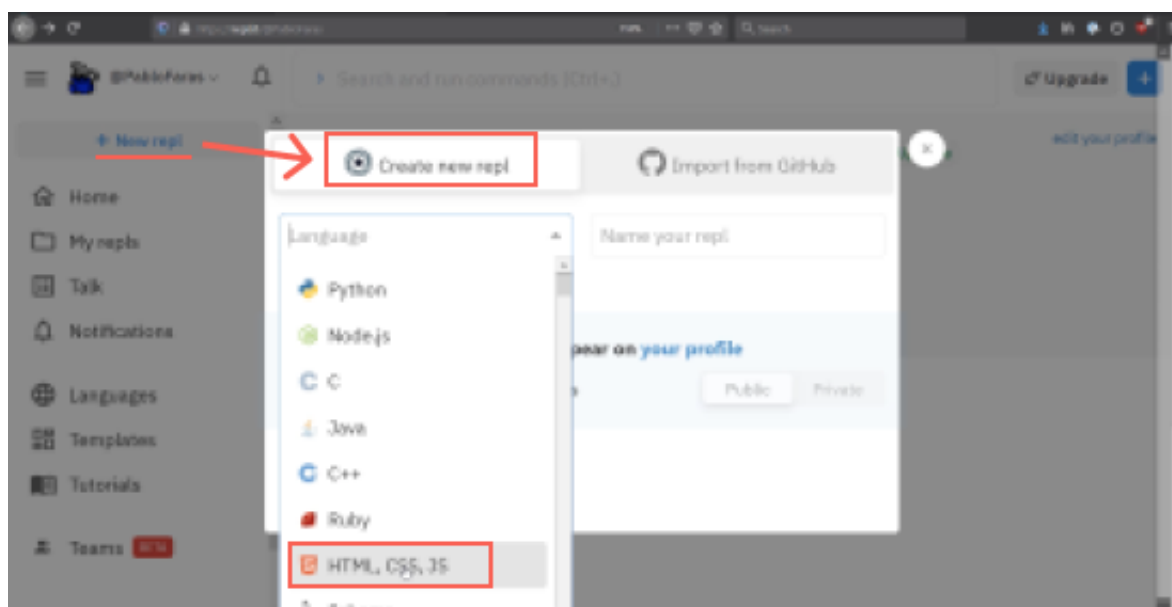
Delete all the code inside **index.html** and replace it with:

```
<script>
    alert('Hello World')
</script>
```
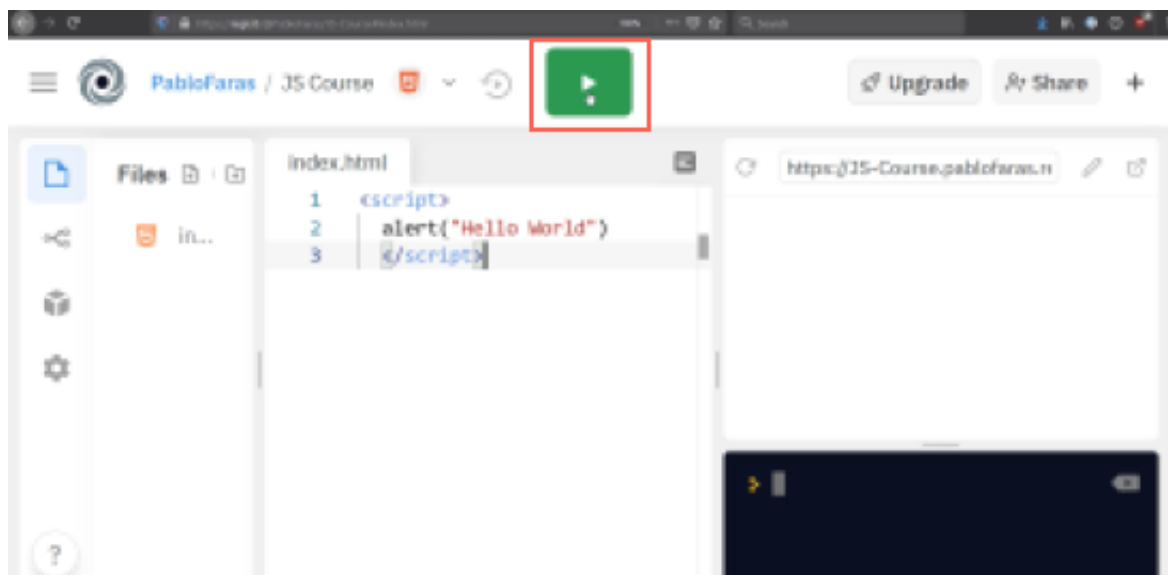

Press the **Run** button and the dialog box with **Hello World** appears in the browser running the **repl.it** website.
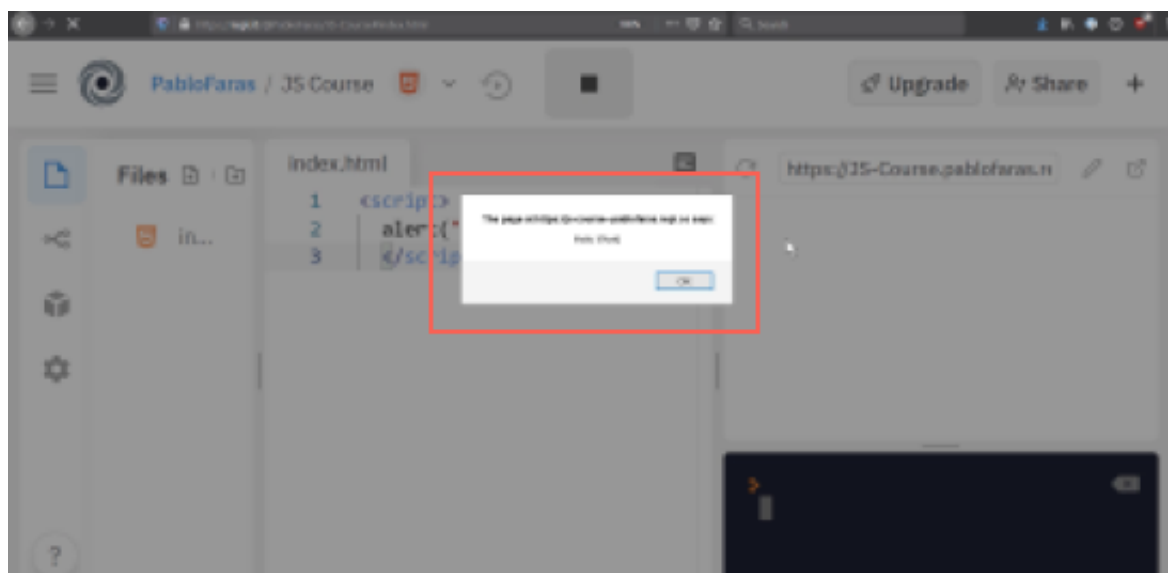
Final Notes

While online coding environments provide a great place for demos, learning, sharing or testing snippets of code, trying out the basics of new languages and more; it is important to mention that if you are serious about programming and want to move to real projects, we recommend using VS Code or another code editor that allows you to setup a computer-based development environment.

Variables are a common construct found in most programming languages. They allow you to store and retrieve data from the memory of your computer. To enable this, you provide variables with a **name** and assign them a **value.** The name can then be used to retrieve or update the value. How you assign a value to a variable differs based on the **type of data** you want to assign. Variables are a key building block of most programming languages including JavaScript. In this lesson we introduce JavaScript's most common (primitive) data types. In future lessons we will see how variables can be used with more complex data structures.
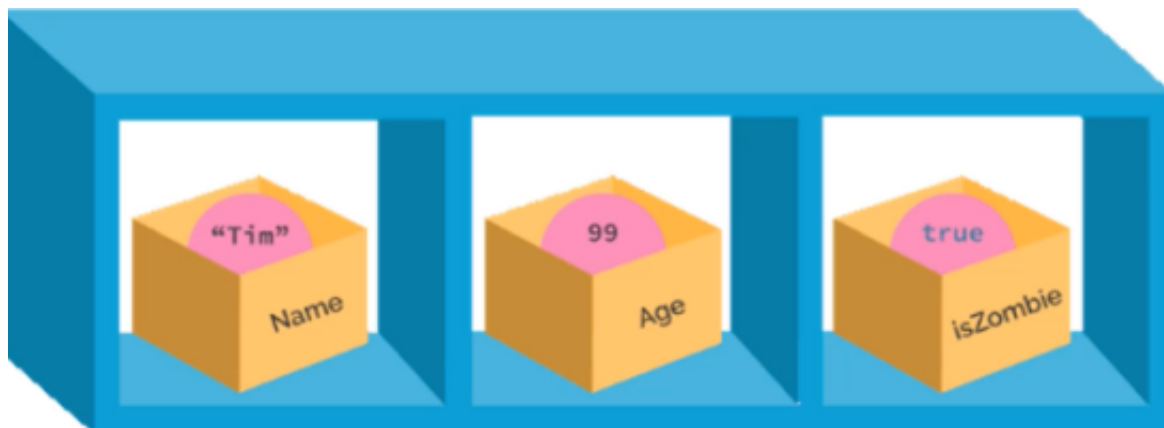
Data Types in JavaScript

Variables in programming languages generally have built-in **data types**. In JavaScript, the four most common data types are:

Data Type	Description
Undefined	We create a variable name without assigning it a value
Boolean	Represents a logical value that is either true or false
Number	Represents any number including integers, decimals and both positive and negative numbers
String	Represents textual data; which is any value inside quotes

The preceding table covers the **Data Types** introduced so far. We will be covering more as the course progresses. A complete list is available from the Mozilla Developer Network (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures)

JavaScript is Dynamically Typed

It's important to note that variables in programming languages can be typed **statically** or **dynamically**. JavaScript is **dynamic**. This means that variables are not directly associated with any particular data type so they can be assigned (and re-assigned) values of all types. For code clarity, it's recommended that you do not change a variable's **data type** unless there is a very specific reason for doing so.

```
<script>
  // double slashes are for comments in JavaScript

  let name;        // name is undefined
  name = true      // name is now a boolean
```

```
name = 10        // name is now a number
name = 'Pilot'   // name is now a string

// using variables in this manner is not recommended
</script>
```

Working with JavaScript

We will keep our code files organized in this course by creating sub-folders for each lesson module. In VS Code, create a sub-folder for **variables**. Inside this folder create a new file named **index.html** just as we did in the previous lesson.
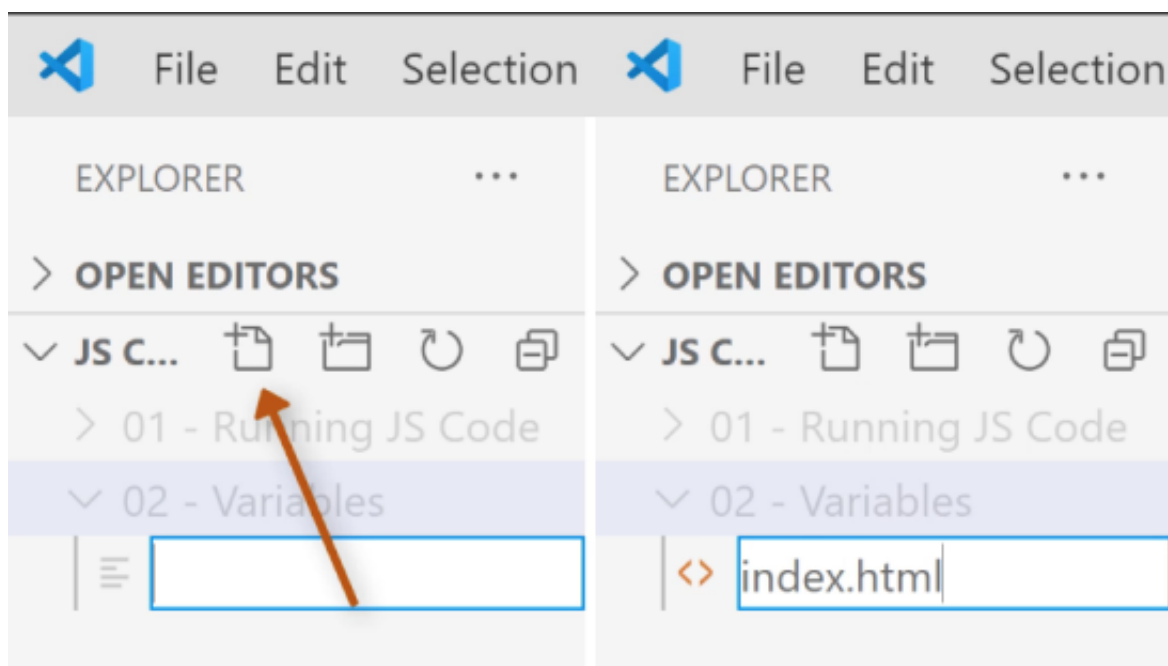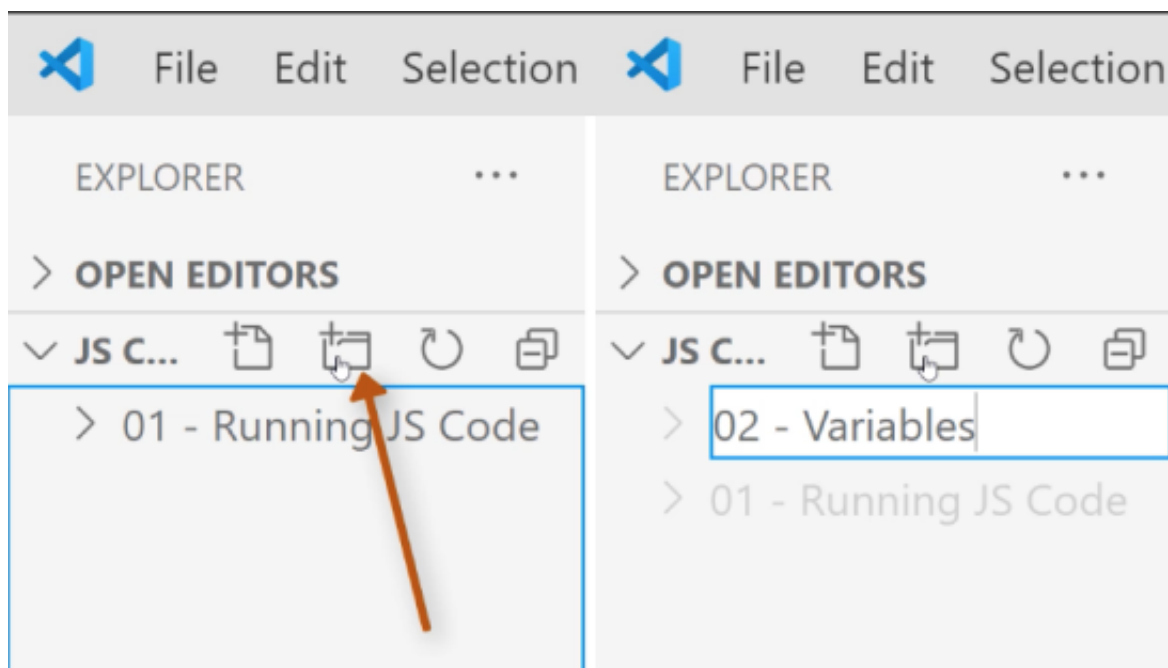
In order to run JavaScript from within a **html** file we must put our code inside
<script></script> tags.

```
<script>
  // write JavaScript code here
</script>
```

One of the advantages of JavaScript is that you can easily run it within the browser (Google Chrome or any modern web browser). It's often a good idea to have your browser visible while you code. Open the newly created **index.html** file in your browser; then open the browser dev tools (right-click to open browser context menu and press **Inspect** then select **Console**). Position VS Code and your browser so both are visible at the same time (you may want to close the VS Code file explorer to provide additional space by clicking on the file explorer icon or using the keyboard shortcut **control+b** on Windows or **cmd+b** on Mac). The **Console** is where we will see variables, code output and errors.
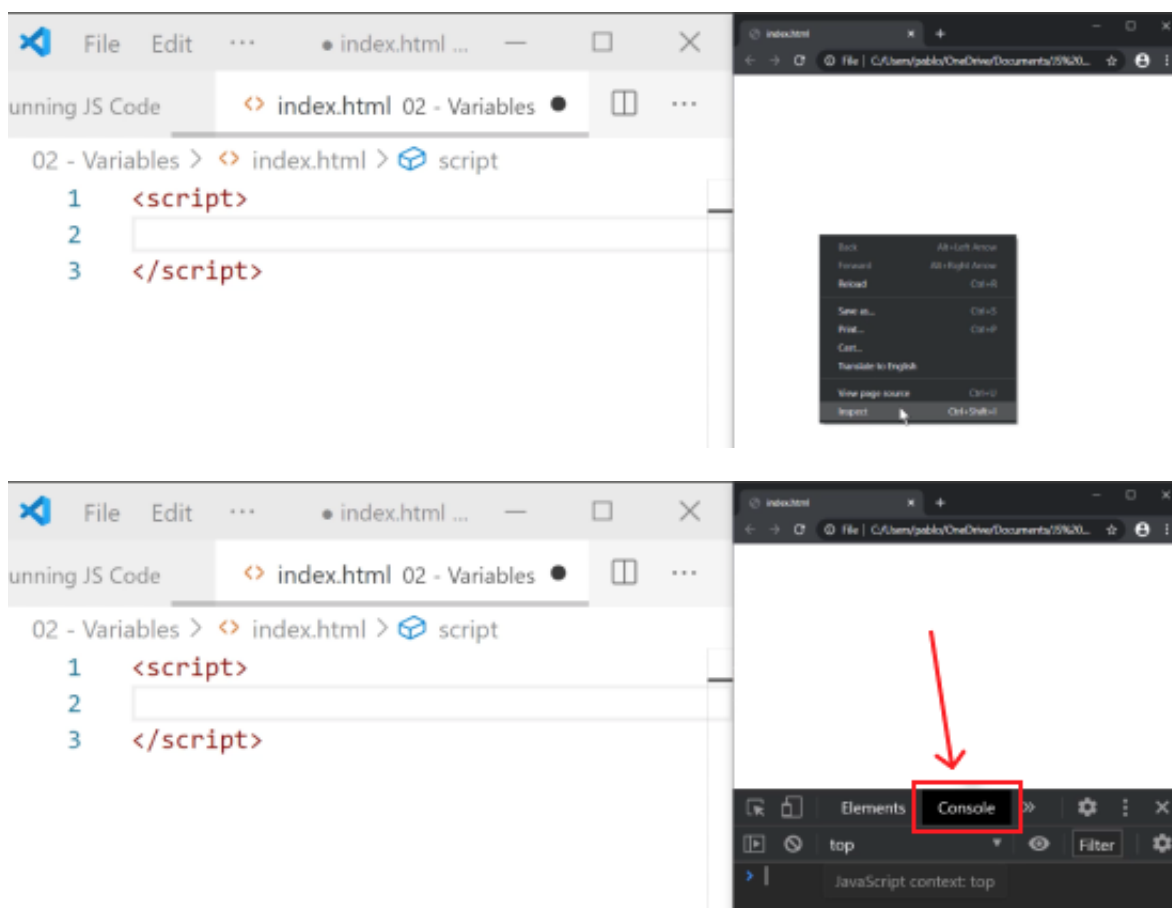
Variable Naming Rules

JavaScript does have few **Rules** to keep in mind regarding **Variable Naming**.

Variable Naming Rules

Names **must start** with a letter or the underscore sign (_)
Names **can** contain letters, numbers and the underscore sign (_)

Names **cannot** be JavaScript reserved **keywords;** for example, let

Variable Declaration and Assignment

When you first name a variable it is called **variable declaration** and when you assign a value to the variable it is called **variable assignment**. The declaration and assignment can happen together or separately. To **declare** a variable we will use the **let** keyword. Also, note that we end each JavaScript statement with a semicolon (**;**). Using the semicolon is not required in JavaScript but it is recommended. To add **Comments** to JavaScript code we use **//** and to display information to the **Console** we use the **console.log()** function.

```
<script>
  // variable declaration
  let age;                     // age is undefined

  // variable assignment
  age = 50;                    // age is now a number

  // show in console
  console.log(age);

  // variable declaration and assignment
  let name = 'Pilot';          // name is a string
  console.log(name);

  // boolean variable (true or false)
  let hasExperience = true;   // hasExperience is a boolean
  console.log(hasExperience);

  // number
  let examResult = 50.5;      // examResult is a number
  console.log(examResult);

  examResult = 100;           // examResult is a new number
  console.log(examResult);

  // not recommended
  examResult = 'Some Text'; // examResult is now text
  console.log(examResult);

  // undefined
  let something1;             // something1 is undefined
  console.log(something1);
</script>
```

All the information we passed to the **console.log()** function is displayed in the **Console**. Refresh the page to display updated code changes.
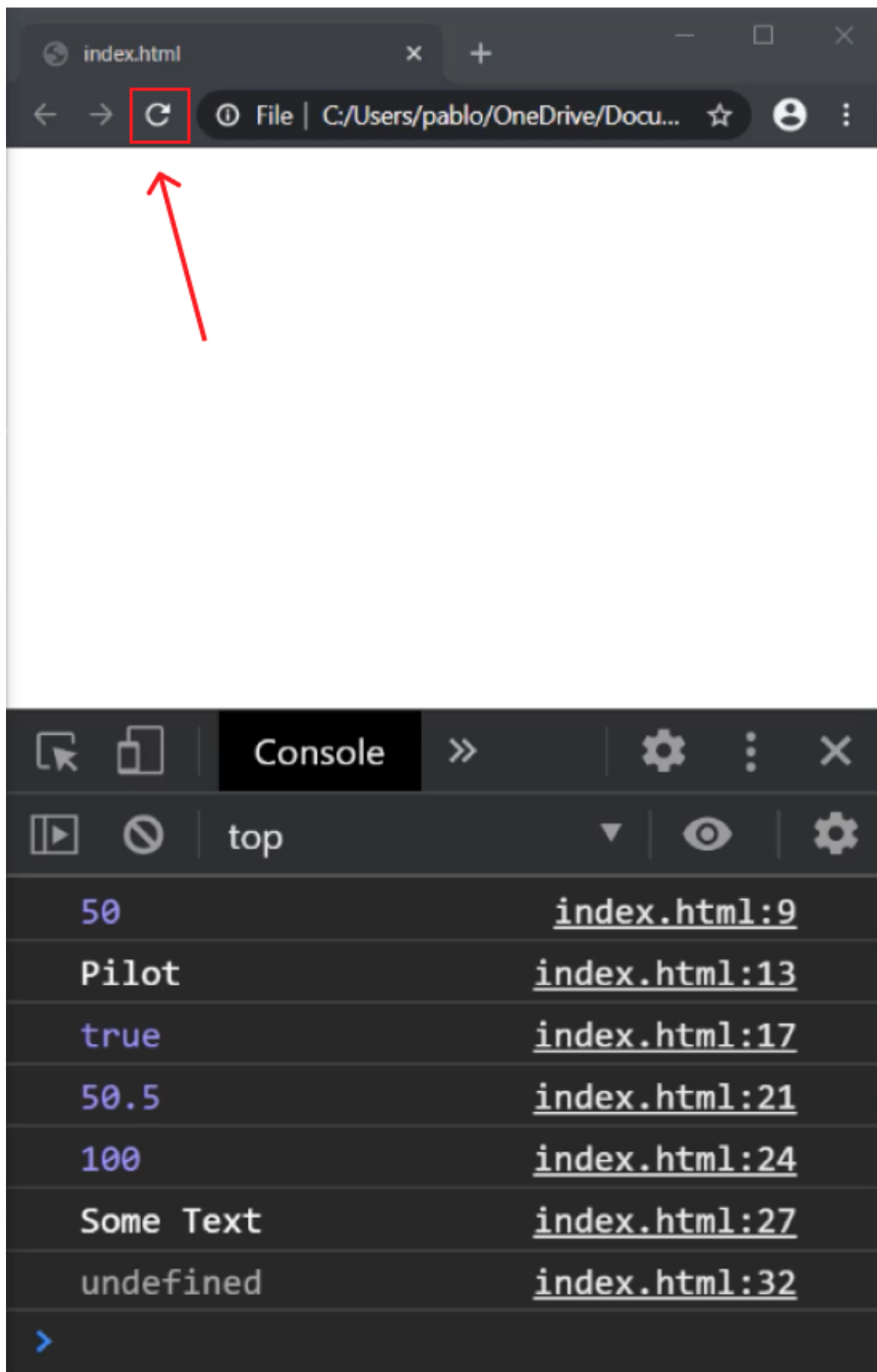
While you can update the values of variables as shown above, one important point about declaring variables with the **let** keyword is that you cannot re-declare the same variable name. If we tried to re-declare **examResult** from the previous code we would get the following error in the console (this error would occur whether you assigned a value or not).
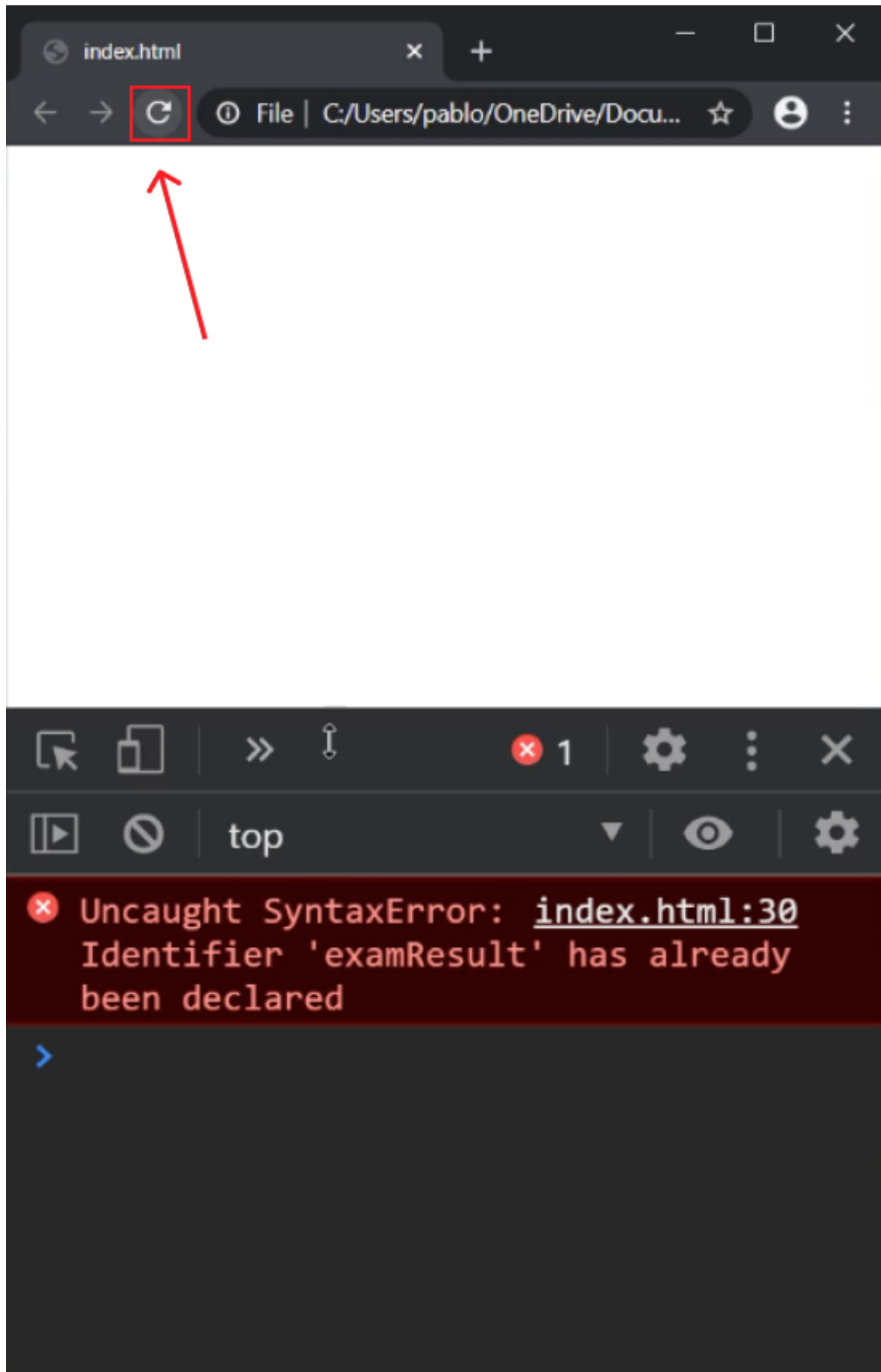
In previous lessons we wrote all our JavaScript code in the **index.html** inside **<script></script>** tags. For this lesson, create a new folder for this **Arithmetic Operations** lesson module. Inside the folder create an **index.html** as we have in previous lessons. In this lesson, we will follow a more common practice which is linking to an **external JavaScript file (script.js)**. The **.js** extension is what identifies this file as a JavaScript file.

```html
<!-- index.html - BEFORE -->
<script>
  // JavaScript code goes here
</script>
```

Use the **src** attribute of the **<script></script>** tag to identify the external JavaScript reference.

```html
<!-- index.html - AFTER (external JavaScript file) -->
<script src="script.js"></script>
```
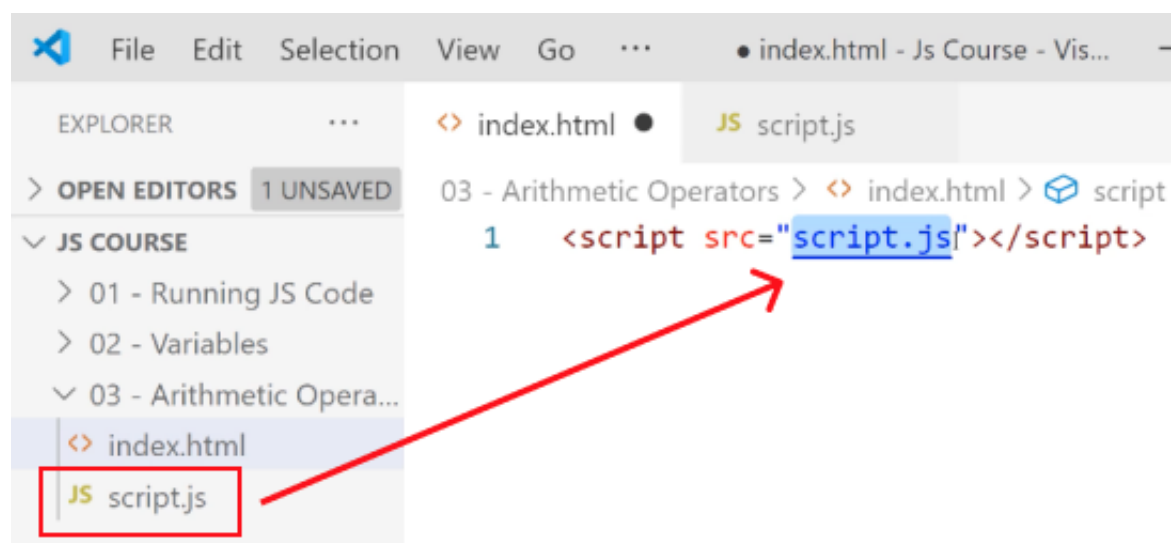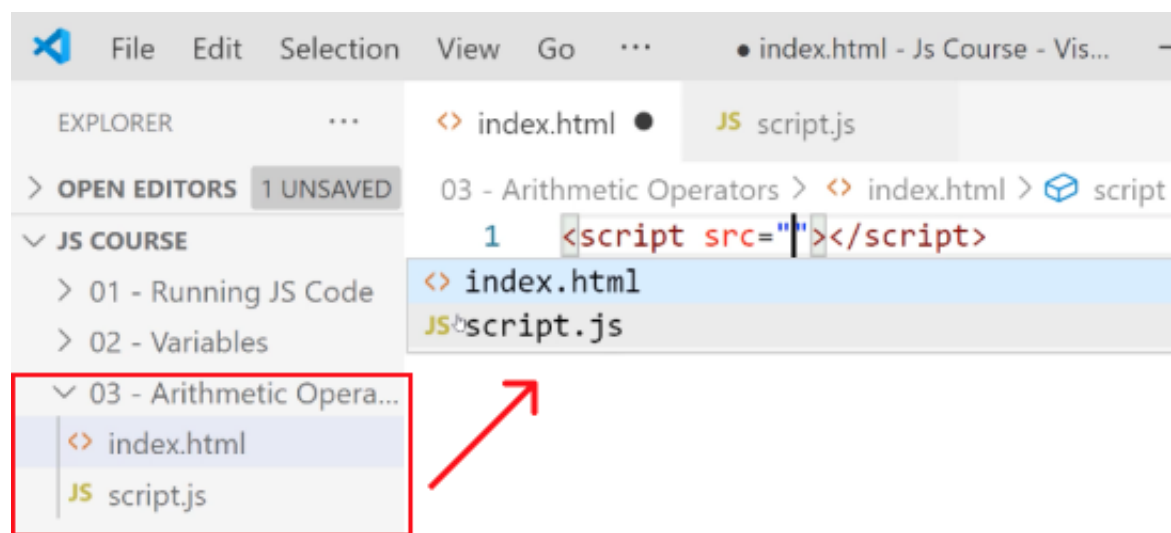

Arithmetic Operations

Operator	Description	Example
Addition (**+**)	Adds two numbers together	10 + 2 (returns 12)
Subtraction (**-**)	Subtracts the right number from the left	10 - 5 (returns 5)
Multiplication (*****)	Multiplies two numbers together	10 * 2 (returns 20)
Division (**/**)	Divides the left number by the right	10 / 2 (returns 5)
Modulus (**%**)	Returns the remainder left over after you've divided the left number into a number of integer portions equal to the right number	8 % 3 (returns 2, as three goes into 8 twice, leaving 2 left over)

Addition

The following **Addition** examples assign results to variables and log the results to the console.

```
// addition
let a = 1 + 1;
console.log(a);

let b = 10;
let c = a + b;
console.log(c);

//b = b + 1;
b += 1;
console.log(b);
```

Code Step	Explanation
let a = 1 + 1;	Sets a to the result of 1 + 1. The new value of a is 2
console.log(a);	Prints 2 in the console
let b = 10;	Sets b equal to 10
let c = a + b;	Sets c equal to result of 2 + 10. The new value of c is 12
console.log(c);	Prints 12 in the console
b += 1;	Increases value of b by 1. Same result as the alternative b = b + 1;. The variable b starts at 10 and is then increased by 1 so the new value for b is 11
console.log(b);	Prints 11 in the console.

Subtraction

The following **Subtraction** examples assign results to variables and log the results to the console.

```
// subtraction
let x = 10 - 5;
console.log(x);

let y = b + x;
```

```
console.log(y);

x = x - 1;
// x -= 1;
console.log(x);
```

Code Step	Explanation
let x = 10 - 5;	Sets x to the result of 10 - 5. The new value of x is 5
console.log(x);	Prints 5 in the console
let y = b + x;	Sets y equal to b + x (11 + 5). The new value of y is 16
console.log(y);	Prints 16 in the console
x -= 1;	Decreases value of x by 1. Same result as the alternative x = x - 1;. The variable x starts at 5 and is then decreased by 1 so the new value for x is 4
console.log(x);	Prints 4 in the console.

Multiplication

The following **Multiplication** examples assign results to variables and log the results to the console.

```
 // multiplication
let unitPrice = 2;
let units = 10;
let total = unitPrice * units;
console.log(total);
```

Code Step	Explanation
let unitPrice = 2;	Sets unitPrice to 2
let units = 10;	Sets units to 10
let total = unitPrice * units;	Sets total equal to unitPrice * units; (2 * 10). The new value of total is 20
console.log(total);	Prints 20 in the console

Division

The following **Division** examples assign results to variables and log the results to the console.

```
// division
let n = 10;
let result = n / 2;
console.log(result);
```

Code Step	Explanation
let n = 10;	Sets n to 10
let result = n / 2;	Sets result equal to n / 2; (10 / 2). The new value of result is 5
console.log(result);	Prints 5 in the console

Modulus

The following **Modulus** examples assign results to variables and log the results to the console.

```
let r = 5 % 2;
console.log(r);
```

Code Step	Explanation
let r = 5 % 2;	Sets r equal to 5 % 2; (5 modulus 2). The new value of r is 1. Returns 1 because 2 goes into 5 twice, with 1 left over
console.log(r);	Prints 1 in the console

Challenge

The **Challenge** is to use the information we are given and knowledge we've gained in this lesson to calculate the total weight of a **ship** including **fuel, food** and **passengers**.

```
// Challenge
// Calculate the total weight of the ship

let baseWeight = 10000;
let foodWeight = 100;
let passengerWeight = 10;

let fuelUnitWeight = 2; // weight of  unit of fuel
let fuelUnits = 100; // total units of fuel we need
```

Tip: when you have a lot of information to calculate you can group those calculations, storing the result of each sub-total into variables that you can then add together to get the final total. For example, in this challenge it would be helpful to calculate the **fuelWeight** (which is currently not known) before determining the total weight.

Solution

The **Solution** is to use the information we are given and knowledge we've gained in this lesson to calculate the total weight of a ship. There are many ways to solve most challenges. By coding along as we go and trying the challenges it truly enhances the learning experience. One way to solve this challenge is by calculating the **fuelWeight** (which is not initially known) then adding that weight to other weights we are given to determine the total weight.

```
// Solution
// Calculating the total weight of the ship

let baseWeight = 10000;
let foodWeight = 100;
let passengerWeight = 10;

let fuelUnitWeight = 2; // weight of  unit of fuel
let fuelUnits = 100; // total units of fuel we need

// Step 1 - calculate the fuel weight
let fuelWeight = fuelUnitWeight * fuelUnits;

// Step 2 - calculate the total weight
let totalWeight = baseWeight + foodWeight + passengerWeight + fuelWeight;
```

```
// Step 3 - log the result to the console
console.log(totalWeight);
```

Code Step	Explanation
let fuelWeight = fuelUnitWeight * fuelUnits;	Sets fuelWeight equal to fuelUnitWeight * fuelUnits; (2 * 100). The new value of fuelWeight is 200
let totalWeight = baseWeight + foodWeight + passengerWeight + fuelWeight;	Sets totalWeight equal to addition of our given weights plus our calculated fuelWeight (10000 + 100 + 10 + 200). The new value of totalWeight is 10310
console.log(totalWeight);	Prints 10310 in the console

In the previous lesson, we introduced linking to an **external JavaScript file (script.js)**. The **.js** extension is what identifies a file as a JavaScript file. This is the approach we will use in this and all future lessons. Setup this lesson as we've done previously by creating a new lesson folder for this **Conditional Statements** module. Inside the folder create the two standard files: **index.html** (our webpage) and **script.js** (our external JavaScript file). Recall, we use the **src** attribute of the **<script></script>** tag to identify the external JavaScript reference.

```
<!-- index.html (reference external JavaScript file) -->
<script src="script.js"></script>
```

We complete our standard setup by opening **index.html** in your browser of choice and having that browser visible next our editor. In the browser, open the browser developer tools (right-click | context menu | choose Inspect | select Console) and we're ready to learn about **Conditional Statements**.
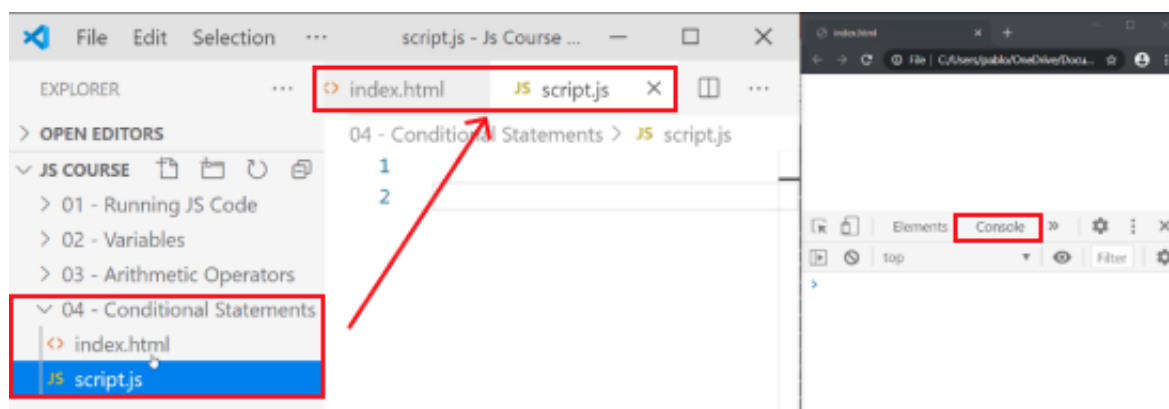
IF Statement

Conditional Statements have three possible options (if | else if | else) but only the **IF** portion is required.
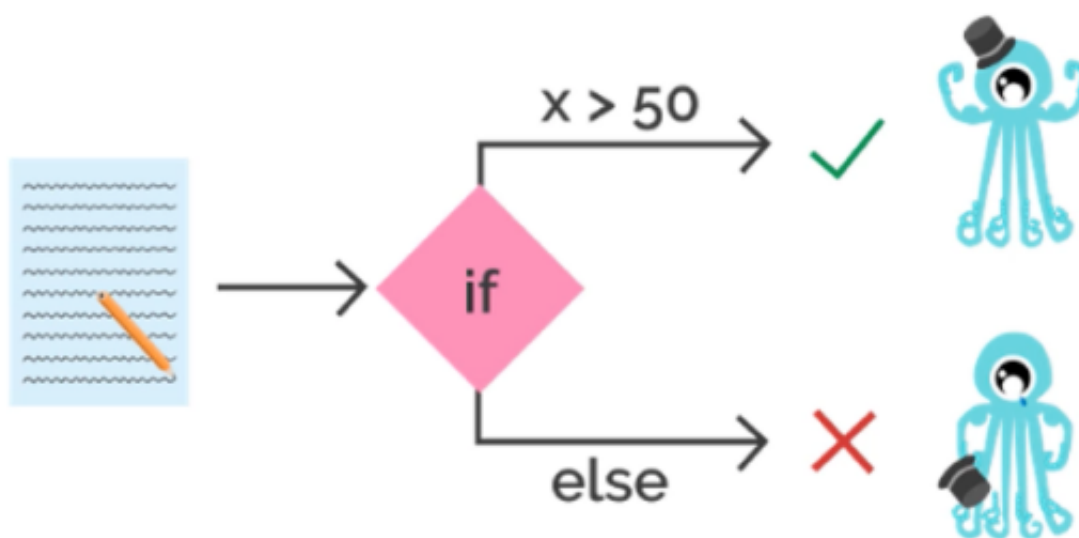
A simple **IF** statement has three parts:

```
if(conditionTrueFalse) {
  /* JavaScript logic runs when condition is true */
}
```

Part	Description
if	The if statement begins with the **IF** keyword and is **required**. There can only be one if within a **conditional block**
(conditionTrueFalse)	The **if** keyword is followed by **parens** that contain a **conditional statement**. This is an expression that evaluates to either **true** or **false**. For example: if(mark > 50) { /* JS Logic */}. In this case if **mark** is greater than **50** then the statement evaluates to **true**
{ /* run JavaScript logic */ }	When the **condition** evaluates to **true** the JavaScript logic

ELSE IF Statement

The **ELSE IF** allows an additional conditional check if the primary **IF** statement returns **FALSE**. You can have as many **ELSE IF** statements as you like follow the initial **IF** statement. As soon as any **if** or **else if** evaluates to true no additional statements containing **IF** are evaluated.

```
if(conditionFALSE) {
  /* JavaScript logic does not run because condition is false */
}

else if(conditionFALSE) {
  /* JavaScript logic does not run because condition is false */
}

else if(conditionTRUE) {
  /* JavaScript logic runs because condition is true */
}

else if(conditionTRUE) {
  /* JavaScript logic does not run because previous `else if` was `true` */
}
```

ELSE Statement

The **ELSE** statement is optional and allows logic to run if all **IF** and **ELSE IF** statements evaluate to **FALSE**. You can have only one **ELSE** statement and there must be a preceding **IF** statement.

```
if(conditionFALSE) {
  /* JavaScript logic does not run because condition is false */
}

else {
  /* JavaScript logic runs when all if and else if statements evaluate to false */
}
```

Example

As an example, consider logic in a game. We can use **Conditional Statements** to control the flow of our game or app based on certain conditions. In this example, consider the variable **direction**. The value of direction could come from user input such as keyboard, mouse or touch. Let's look at what happens as the variable **direction** changes.

Notice the use of the equal signs. We use one equal sign **=** to assign a value to a variable: let direction = 'left'; and we use two equal signs **==** to compare values and variables: **if(direction == 'left') {}**

```
let direction = 'diagonal';

if(direction == 'left') {
    console.log('red planet');
}

else if(direction == 'right') {
    console.log('purple planet');
}

else if(direction == 'backwards') {
    console.log('back home');
}

else {
    console.log('ship crashes');
}
```

Variable Direction	Code Path	Console Output
'left'	if	red planet
'right'	else if	purple planet
'backwards'	else if	back home
'diagonal' or anything-else	else	ship crashes

A comparison operator compares its operands and returns a **Boolean** value based on whether the comparison result is true or false.

Common Comparison Operators

The following table contains six common **Comparison Operators:**

Operator	Description
==	Equality operator
!=	Inequality operator
>	Greater than operator
>=	Greater than or equal operator
<	Less than operator
<=	Less than or equal operator

Equality Example

```javascript
let item = 'engine';

if(item == 'engine') {
    console.log('engine!');
}

console.log('value of item is ', item);
```

Value of item	Code Path	Console Output
'engine'	if	engine!
'engine'	console.log()	value of item is engine

Inequality Example

```javascript
let item = 'window';

if(item != 'engine') {
    console.log('not engine!');
}

console.log('value of item is', item);
```

Value of item	Code Path	Console Output
'window'	if	not engine!
'window'	console.log()	value of item is window

Relational Example

```javascript
let score = 5;

if(score >= 60) {
    console.log('pass');
}
else if (score < 10) {
    console.log('meet the teacher');
}
else {
    console.log('not pass');
```

```
}
```

Value of score	Code Path		Console Output
90	if	pass	
60	if	pass	
59	else if	not pass	
10	else if	not pass	
9	else	meet the teacher	
5	else	meet the teacher	

Assigning Comparison Result to Variable

Another option is to assign the result of the **comparison** to a variable, then evaluate the variable inside an **IF** statement. This is particularly useful if the result has any calculations or if you are passing the result to another part of your app.

```
let item = 'engine';

let isEngine = item == 'engine';
console.log(isEngine);

if(isEngine) {

}
```

Code Step	Explanation
let item = 'engine';	Sets item equal to **String** 'engine'
let isEngine = item == 'engine';	Sets isEngine equal to **Boolean** true since the variable item ('engine') is equal to the **String** 'engine'
console.log(isEngine);	Prints true in the console based on the result of the previous step
if(isEngine) { }	Code inside the curly braces ({ }) is executed if the isEngine variable is equal to **Boolean** true

Challenge

The **Challenge** is to use the information we are given and knowledge we've gained in this lesson to determine if a customer has enough money (**balance**) to buy an item (**itemPrice**). If they do, sell the item and update the **balance**. If not, indicate this fact by logging a message to the console.

Solution

There are many ways to solve most challenges. By coding along as we go and trying the challenges we enhance our learning experience. The following is one way to solve this challenge:

```
// Challenge

// Balance BEFORE first purchase is set to 1100
let balance = 1100;
// Balance AFTER first purchase is set to 100
let itemPrice = 1000;

// 1. check balance
// 2. if they buy, update balance

if(balance >= itemPrice) {
```

```
    // update balance
    // balance = balance - itemPrice;
    balance -= itemPrice;
    console.log('item purchased');
    console.log(balance);
}
else {
    console.log('not enough balance');
}
```

First execution of this code:

Code Step	Explanation
let balance = 1100;	Sets balance equal to 1100
let itemPrice = 1000;	Sets itemPrice equal to 1000
if(balance >= itemPrice)	Checks if balance is greater than or equal to itemPrice (1100 >= 1000 returns **true**)
balance -= itemPrice;	Sets balance equal to balance - itemPrice; (1100 - 1000). The new value of balance is 100
console.log('item purchased');	Prints item purchased in the console
console.log(balance);	Prints 100 in the console

Second execution of this code (**balance** is now **100**):

Code Step	Explanation
if(balance >= itemPrice)	Checks if balance is greater than or equal to itemPrice (100 >= 1000 returns **false**)
console.log('not enough balance');	Prints not enough balance in the console

In the last lesson, we learned about **Comparison Operators** and how to use them inside **IF** and **ELSE IF** statements. In this lesson we learn about **Boolean Logic** which allows us to evaluate more than one condition in a single statement using three **Boolean Operators** (**AND, OR, NOT**). This applies to not just to **IF** and **ELSE IF** but also to the **assignment** of a comparison result to a **Boolean Variable**.

Example

Boolean Logic can help us determine outcomes in our apps based on data and logic. In this example scenario we have to determine if our spacecraft can make it to the planet before running out of fuel. The conditions are in the following image and our known data is in the code below.

Condition	What We Know
Distance > 200	Too far, won't make it
Distance between 200 and 100	Need at least 100 fuel units
Distance < 100	Need at least 25 fuel units
Engine Damaged	Won't make it regardless of any other condition

```javascript
let distance = 250;
let fuel = 100;
let distanceCondition = distance <= 200 && distance >= 100;
let isEngineFunctioning = true;

// FIRST Evaluation
if(!isEngineFunctioning || distance > 200 ) {
  console.log('wont make it');
}

// SECOND Evaluation (check if FIRST is false)
else if(distanceCondition && fuel >= 100) {
  console.log('you will make it');
}

// THIRD Evaluation (check if FIRST and SECOND false)
else if(distance < 100 && fuel >= 25) {
    console.log('you will make it');
}
```

We are evaluating conditions inside **IF** and **ELSE IF** statements which means once a condition

is **true** no other conditions are checked.

Evaluation	Comparison Result	Outcome
FIRST	true	Prints 'wont make it' in the console
FIRST	false	Check **SECOND Evaluation**
SECOND	true	Prints 'you will make it' in the console
SECOND	false	Check **THIRD Evaluation**
THIRD	true	Prints 'you will make it' in the console
THIRD	false	No code blocks run. Nothing printed to console

Logical NOT (!)

The first check in our example is to determine if our engine if functioning. We do this check first because without our engine functioning there is no way to make it to our destination. We set a variable **isEngineFunctioning** and set its initial value to **true**. We then want to check if the engine is NOT working so we use the **negation** (!) symbol which is used with **Boolean** values to return its opposite.

```
if(!isEngineFunctioning ) {
  console.log('isEngineFunctioning - NO');
}
```

Value of isEngineFunctioning	Condition Check	Condition Value
true	!isEngineFunctioning	false
false	!isEngineFunctioning	true

Logical AND (&&)

All conditions must be **true** for **result** of expression to be **true**. Consider the **THIRD Evaluation** from example code above.

```
let distance = 50;
let fuel = 100;

if(/*condition1*/) {}
else if(/*condition2*/) {}

// THIRD Evaluation (check if FIRST and SECOND false)
else if(distance < 100 && fuel >= 25) {
    console.log('you will make it');
}
```

Value of distance	Code Part	Explanation
50	distance < 100	First condition (50 < 100) evaluates to true so condition after the AND (&&) is evaluated
50	fuel >= 25	Second condition (100 >= 25) evaluates to true. Since both conditions evaluated to true the condition value is true and the code inside the curly braces ({ }) is executed

```
let distance = 250;
let fuel = 100;
```

```
if(/*condition1*/) {}
else if(/*condition2*/) {}

// THIRD Evaluation (check if FIRST and SECOND false)
else if(distance < 100 && fuel >= 25) {
    console.log('you will make it');
}
```

Value of distance	Code Part	Explanation
250	distance < 100	First condition (250 < 100) evaluates to false so condition after the AND (&&) is **NOT** evaluated since all conditions must be true for condition value to be true. The condition value is therefore false and the code inside the curly braces ({ }) is NOT executed

Logical OR (||)

At least one of the conditions must be **true** for **result** of expression to be **true**. If the first condition evaluates to **true** then **true** is immediately returned as the condition value. If the first condition evaluates to **false** then the second condition is evaluated and its result becomes the condition value.

In our example, there are two conditions that we know will prevent us from reaching our destination. We discussed one above (the negation of **isEngineFunctioning**). Since a second condition will also, on its own, prevent us from reaching our destination we use the **Boolean OR (||) Operator** to determine as part of our **FIRST Evaluation** if we won't make it.

```
let distance = 100;
let isEngineFunctioning = true;

// FIRST Evaluation
if(!isEngineFunctioning || distance > 200 ) {
  console.log('wont make it');
}
```

Value of isEngineFunctioning	Code Part	Explanation
true	!isEngineFunctioning	The first condition includes the **negation** (!) symbol. This means that since isEngineFunctioning is true its opposite false is returned and the second condition (distance > 200) is evaluated. Since (100 > 200) evaluates to false the condition's result is false and won't make it is logged to the console

```
let distance = 100;
let isEngineFunctioning = false;

// FIRST Evaluation
if(!isEngineFunctioning || distance > 200 ) {
  console.log('wont make it');
}
```

Value of isEngineFunctioning	Code Part	Explanation
false	!isEngineFunctioning	The first condition includes the **negation** (!) symbol. This means that since isEngineFunctioning is false its opposite true is returned. This means that true is returned as the condition value, the second condition (distance > 200) is NOT evaluated since only one condition needs to be true, and won't make it is logged to the console

Boolean Variables

We can also use a **Boolean Variable** to hold the result of a condition. In our example, **distanceCondition**.

```
let distance = 100;
let distanceCondition = distance <= 200 && distance >= 100;
```

Value of distance	Code Part	Explanation
100	distance <= 200	First condition (100 <= 200) evaluates to true so condition after the AND (&&) is evaluated
100	distance >= 100	Second condition (100 >= 100) evaluates to true. Since both conditions evaluated to true the variable distanceCondition is set to true

```
let distance = 250;
let distanceCondition = distance <= 200 && distance >= 100;
```

Value of distance	Code Part	Explanation
250	distance <= 200	First condition (250 <= 200) evaluates to false so condition after the AND (&&) is **NOT** evaluated and the variable distanceCondition is set to false

You can then use the variable **distanceCondition** as part of another conditional check. Consider the **SECOND Evaluation** from above: distanceCondition && fuel >= 100. If the variable **distanceCondition** is **true**, then the second condition (fuel >= 100) is evaluated;

otherwise the condition value is immediately **false**.

Functions are a fundamental building block of many programming languages including JavaScript. A function in JavaScript is like a recipe in that it contains a set of instructions but does not execute those instructions until the function is executed (until the baking of the cake begins). Functions allow for improved code organization and less repetition as they are self-contained blocks of code that can be called as many times as needed. Functions can be written as a **Function Declaration** or **Function Expression**.

Function Parameters and Outputs

Functions optionally accept one or more inputs (called parameters) and optionally return an output. They are often most powerful when they return an output based on the input.
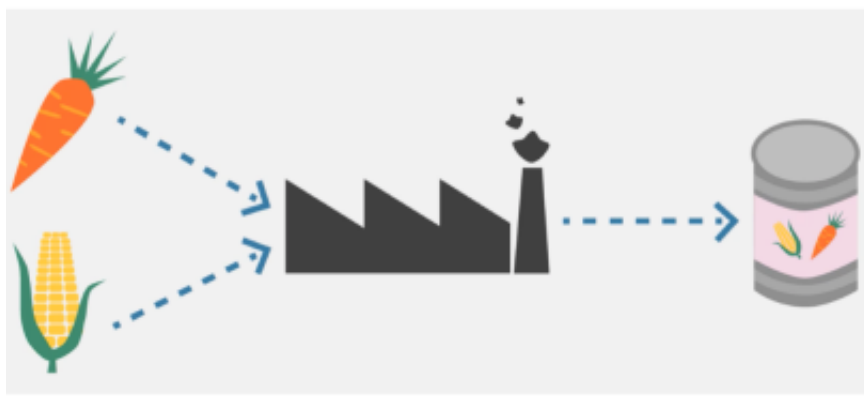
Function Declaration

The **Function Declaration** consists of the **function** keyword, followed by a **name** and an optional list of **parameters** to the function, enclosed in **parentheses** and separated by commas. Next, the **body** of the function contains any number of JavaScript statements that define the function, enclosed in curly brackets ({...})

```
function hourToMinutes(hours) {
  let result = hours * 60;
  console.log(result);
  return result;
}
```

Code Part	Example	Description
Function Keyword	function	Declares code block as a JavaScript function
Function Name	hourToMinutes	Gives function a name that can

		be used to call it
Parentheses	(hours, p2, p3)	The parentheses are required. The zero to many comma separated parameters inside the parentheses are optional
Function body	{ let result = hours * 60; }	The JavaScript statements that define the function, enclosed in curly brackets
Return Statement	return result;	The optional return statement passes output back to the code that called the function. As soon as a return executes the function execution ends

Function Declaration Example

As an example consider a function that converts hours to minutes. Notice that we call the same function; executing the same logic but with different inputs receiving back corresponding outputs.

```
// a function to convert hours to minutes
// 1 hour = 60 minutes
function hourToMinutes(hours) {
  let result = hours * 60;
  console.log(result);
  return result;
}

// execute the function
let a = hourToMinutes(10);
let b = hourToMinutes(20);
```

Execute the function using let a = hourToMinutes(10); which runs the function passing 10 as a parameter and stores the result in variable a

Code Part	Explanation
function hourToMinutes(hours) {	Function accepts the parameter value of 10 and executes the code inside the curly brackets ({ })
let result = hours * 60;	Sets result equal to hours * 60; (10 * 60). The new value of result is 600
console.log(result);	Prints 600 in the console
return result;	Function returns the result (600) to the code that called the function

Execute the function again using let b = hourToMinutes(20); which runs the function passing 20 as a parameter and stores the result in variable b

Code Part	Explanation
let b = hourToMinutes(20);	Executes function passing 20 as a parameter and stores the result (1200) in variable b
function hourToMinutes(hours) {	Function accepts the parameter value of 20 and executes the code inside the curly brackets ({ })
let result = hours * 60;	Sets result equal to hours * 60; (20 * 60). The new value of result is 1200

console.log(result);	Prints 1200 in the console
return result;	Function returns the result (1200) to the code that called the function

Function Expression

Another option in JavaScript is to create a **Function Expression** which is when you declare a function and assign it to a variable. The function can then be called using the variable name. It's important to note that functions in JavaScript are just another **Data Type** so variables containing functions can be passed as parameters, for example, just like variables that are strings, booleans or numbers.

Function Expression Example

When functions are created as a **Function Expression** they may or may not have a name. Fuctions without names are called **anonymous**. Notice in the example below there is no name following the function keyword. The function is executed by using the variable name.

```
// function expression to convert days to hours

let dayToHours = function(days) {
  return days * 24;
};

let c = dayToHours(1);
console.log(c);
```

Execute the function using let c = dayToHours(1); which runs the function passing 1 as a parameter and stores the result in variable c

Code Part	Explanation
let dayToHours = function(days) {};	Sets the variable dayToHours equal to a function
return days * 24;	When the function is executed, JavaScript first multiplies the days parameter by 24 (1 * 24) then returns the result (24) to the code that executed the function. In this example the result is assigned to the variable c and the result (24) is printed in the console

Notice that function expressions end with a semicolon while function declarations do not.

Shop Example

Imagine a scenario where you want to create a shop. The following table contains our list of requirements:

Requirements
Process each order as it comes in
Check whether there is enough stock
Reduce stock when item purchased
Increase account balance when item purchased
Need to be able to easily repeat procedure

We start by declaring variables and assigning initial values based on what we know. Next, we setup a function (Function Declaration in this example but a Function Expression could also be used). Our function accepts one parameter (**quantity**) and checks if we have enough stock. If not, we log a message to the console. If we do have enough stock then the code inside the if(quantity <= stock) {...} block is executed.

```javascript
// variables declaration
let balance = 100;
let stock = 50;
let price = 5;

function sellItem(quantity) {
  // check if we have stock
  if(quantity <= stock) {
    // reduce stock, increase balance
    // stock = stock - quantity;
    stock -= quantity;

    // balance = balance + price * quantity;
    balance += price * quantity;

    console.log('purchase completed', balance, stock);
  }
  else {
    console.log('not enough stock');
  }
}

// execute the function
sellItem(10);
sellItem(10);
sellItem(10);
```

Execute the **sellItem** function using sellItem(10); which runs the function for the **FIRST** time passing 10 as a parameter.

Code Part	Explanation
if(quantity <= stock) {...}	Evaluates if there is enough stock (10 <= 50>). Evaluates to true so if code block is executed
stock -= quantity;	Sets stock equal to stock - quantity; (50 - 10). The new value of stock is 40
balance += price * quantity;	Sets balance equal to balance + (price * quantity); (100 + (5 * 10)). The new value of balance is 150
console.log('purchase completed', balance, stock);	Prints purchase completed 150 40 in the console

Execute the **sellItem** function again using sellItem(10); which runs the function for the **SECOND** time passing 10 as a parameter.

Code Part	Explanation
if(quantity <= stock) {...}	Evaluates if there is enough stock (10 <= 40>). Evaluates to true so if code block is executed

Code Part	Explanation
stock -= quantity;	Sets stock equal to stock - quantity; (40 - 10). The new value of stock is 30
balance += price * quantity;	Sets balance equal to balance + (price * quantity); (150 + (5 * 10)). The new value of balance is 200
console.log('purchase completed', balance, stock);	Prints purchase completed 200 30 in the console

Execute the **sellItem** function again using sellItem(10); which runs the function for the **THIRD** time passing 10 as a parameter.

Code Part	Explanation
if(quantity <= stock) {...}	Evaluates if there is enough stock (10 <= 30>). Evaluates to true so if code block is executed
stock -= quantity;	Sets stock equal to stock - quantity; (30 - 10). The new value of stock is 20
balance += price * quantity;	Sets balance equal to balance + (price * quantity); (200 + (5 * 10)). The new value of balance is 250
console.log('purchase completed', balance, stock);	Prints purchase completed 250 20 in the console

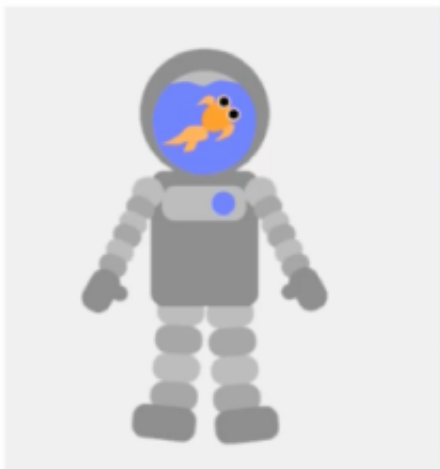Notice that without using **Functions**, we would have to repeat the **sellItem** logic every time we sold an item. This is not practical in the real world so calling the same logic wrapped in a function is an extremely important programming concept.

Objects are data structures that allow you to store multiple properties and more complex entities in a single variable. Think of an object as a collection of properties that have a name and a value.

Name	"John"
Age	99
isZombie	false

In this lesson we learn about **Object Declaration**, **Accessing Object Properties**, **Modify Object Properties** and **Nested Objects**.

Object Declaration

Imagine your are creating a game and need to store information about your player. You could consider all the details and store them as separate variables.

Player Representation
Name
Score
Health
Active
Outfit – color
Outfit – size
Outfit – cost

However, JavaScript provides the **Object Data Type** as a better way to store this type of related data. We can represent the player by declaring an **Object**. To create an object, simply use the curly brackets ({...}) and add a semicolon at the end.

```
// declare the object
let player = {};

console.log(player);
```

Console Output	Explanation
▶ {}	Empty object with no custom properties. You may also notice __proto__: Object which contains built-in object methods that you can safely ignore for this course

To create properties, simply go inside the curly brackets and add a property name followed by a colon, space and property value.

```
// declare the object
let player = {
  score: 99
};


console.log(player);
```

Console Output	**Explanation**
► {score: 99}	The object is printed in the console. Notice a small arrow (►) to the left of the object. Click this arrow to expand the object and view all its properties and values

To add more properties, simply use a comma to separate them. It's common practice to put each property name/value pair on its own line. Let's enter more of the player information we identified above and observe the output printed to the console.

```
// declare the object
let player = {
  name: 'ABC',
  score: 99,
  isActive: true,
  outfitColor: 'blue',
  outfitSize: 'M'
};


console.log('player object:',player);
```

```
player object:
? {name: "ABC", score: 99, isActive: true, outfitColor: "blue", outfitSize: "M"}
  isActive: true
  name: "ABC"
  outfitColor: "blue"
  outfitSize: "M"
  score: 99
```

Notice that there are multiple properties we want to keep associated with the player's **outfit**. JavaScript allows us to group associated pieces of information inside an object by using a **Nested Object**.

Nested Objects

When you have objects that contain other objects as properties it is referred to as **Nested Objects**. The player's **outfit** is a perfect use case for this option. Let's refactor the **player** object to include **outfit** as a **nested object** with additional properties and values.

```
// declare the object
let player = {
  name: 'ABC',
  score: 99,
  isActive: true,
```

```
  outfit: {
    color: 'blue',
    size: 'M',
    cost: 100,
    isNew: true
  }
};

console.log('player object:',player);
```

Notice that the **outfit** property is now represented as an object (outfit: {...}).

```
player object:
?? {name: "ABC", score: 99, isActive: true, outfit: {…}}
```

Expand the **player** object and notice that the **outfit** property can also be expanded.

```
player object:
? {name: "ABC", score: 99, isActive: true, outfit: {…}}
    isActive: true
    name: "ABC"
  ?? outfit: {color: "blue", size: "M", cost: 100, isNew: true}
    score: 99
```

Expand the **outfit** object to see its properties and values.

```
player object:
? {name: "ABC", score: 99, isActive: true, outfit: {…}}
    isActive: true
    name: "ABC"
  ? outfit:
      color: "blue"
      cost: 100
      isNew: true
      size: "M"
    score: 99
```

Accessing Object Properties

To access properties after you have declared the object you can simply type the name of the **object** then **dot** and the name of the **property**. We can demonstrate that by console logging any property of the **player** object.

```
console.log(player.name); // logs ABC
```

Another way of achieving this exact same result is with **bracket notation**. Notice that the property name is surrounded by quotes inside the square brackets.

```
console.log(player['name']); // logs ABC
```

We can access **nested** properties following the same syntax. To access the **color** property of **outfit**.

```
console.log(player.outfit.color); // logs blue
console.log(player['outfit']['color']); // logs blue
```

There are certain more advanced use cases where the **bracket notation** is needed but for just starting out with JavaScript the **dot** syntax is often easier to use.

Modify Object Properties

When it comes to modifying properties of an object we following the same approach as we did for accessing properties. Again, you can use either the **dot** or the **bracket** syntax.

```
// modifying an object property
console.log(player.isActive); // logs true
player.isActive = false;
console.log(player.isActive); // logs false
```

```
// modifying a nested property
console.log(player.outfit.color); // logs blue
player.outfit.color = 'red';
console.log(player.outfit.color); // logs red
```

Adding Object Properties

When it comes to adding object properties we follow the same approach as we did for modifying properties. Again, you can use either the **dot** or the **bracket** syntax.

```
// adding new properties to the object
player.health = 100;

console.log('player object:',player);
```

Notice in the console output the property for **health** with its value set to **100**:

```
player object:
? {name: "ABC", score: 99, isActive: false, outfit: {…}, health: 100}
    health: 100
    isActive: false
    name: "ABC"
```

```
? outfit: {color: "red", size: "M", cost: 100, isNew: true}
   score: 99
```

Deleting Object Properties

For the sake of completeness, we will cover how to **delete** a property but this is not done often and should always be done with great care. When it comes to deleting object properties we follow the same approach as we did for modifying properties. Again, you can use either the **dot** or the **bracket** syntax.

```
// deleting a property
delete player.health;

console.log('player object:',player);
```

Notice in the console output that the property for **health** which was visible in the last section is now gone.

```
player object:
? {name: "ABC", score: 99, isActive: false, outfit: {…}}
   isActive: false
   name: "ABC"
 ? outfit: {color: "red", size: "M", cost: 100, isNew: true}
   score: 99
```

A **Method** is a **Function** that lives inside an **Object** in same way other properties do. Instead of holding a value like a String, Number, Boolean, or Nested Object, the property holds a Function as its value. Since Methods live inside an Object we must use a special JavaScript **Context** keyword **(this)** inside the function to access other Object properties. In this lesson, we will use the example of creating a **Virtual Pet** game to demonstrate the concepts of Method **declaration**, **execution**, and **context**. Our game will require certain pieces of information (**State**) to be stored and certain actions (**Methods**) to modify this information. We can use an Object to store both the State and the Methods.

Method Declaration

Methods are Functions declared as an Object property. Let's begin our example by declaring an Object **player** with two properties like we have seen before **health** and **fun**. Next, we will add the first of our three required Methods, **play**. It's important to note that the Method **play** is defined as a property of **player** with a Function assigned as its value but the Function itself is only defined; not executed.

```
// Declare Object player with Method play
let player = {
  health: 100,
  fun: 0,
  play: function() {
    this.fun += 10;
  }
};
```

Context

The purpose of the **play** Method is to increase the property **fun** indicating that our player enjoys playing. If we try to access **fun** directly we get an error in the console telling us that **fun** is not defined. Since Methods live inside an Object we must use a special JavaScript **Context** keyword **(this)** inside the function to access other Object properties. The keyword **this** refers to the Object (**player**) which owns the Method and gives us access to all its properties including **fun** which we can now modify inside the **play** Method.

Method Execution

Methods are executed just like Functions with one important difference. The name of the Object must be used to execute the Function inside it. The example below causes the code to execute displaying the following output in the console.

```
// Declare Object player with Method play
let player = {
  health: 100,
  fun: 0,
  play: function() {
    this.fun += 10;
  }
};

console.log('player object BEFORE:', player);

// Execute Method play on Object player
```

```
player.play();

console.log('player object AFTER:', player);
```

Browser Developer Console output:

```
player object BEFORE:
?? {health: 100, fun: 0, play: ƒ}
player object AFTER:
?? {health: 100, fun: 10, play: ƒ}
```

Notice that the **play** property is represented as a Function (play: ƒ) and the value of **fun** has increased by **10** after executing player.play();

Virtual Pet Player Object Requirements

This lessons example defined two **State** properties to track our player's **health** and **fun**. We also have the requirement to have three **Methods** that either increase or decrease the values of our player's **health** and **fun** when executed. These requirements can be summarized in a table.

Property	Type	Explanation
health	Number	Holds the value associated with the health of our player. Initial value will be 100
fun	Number	Holds the value associated with the level of fun our player is experiencing. Initial value will be 10
play	Method	Function that when executed increases the value of fun by 10
eatApple	Method	Function that when executed increases the value of health by 10
eatCandy	Method	Function that when executed decreases the value of health by 5 while also increasing the value of fun by 5

Virtual Pet Player Object Solution 1

Starting with the **player** Object, including **play** Method from our code above, we now need to add two additional Methods that both involve eating. These Methods can be added as two additional properties in the same manner as **play**.

```
// Declare Object player with Method play
let player = {
  health: 100,
  fun: 0,
  eatApple: function() {
    console.log('eat apple');

    //this.health = this.health + 10;
    this.health += 10;

    console.log(this.health);
  },
  eatCandy: function() {
    this.health -= 5;
    this.fun += 5;
  },
  play: function() {
    this.fun += 10;
  }
};

console.log('player object BEFORE:', player);
player.play();      // health is 100; fun is 10
player.eatApple(); // health is 110; fun is 10
player.eatCandy(); // health is 105; fun is 15
console.log('player object AFTER:', player);
```

Browser Developer Console output:

```
player object BEFORE:
?? {health: 100, fun: 0, play: ƒ, eatApple: ƒ, eatCandy: ƒ}
player object AFTER:
?? {health: 105, fun: 15, play: ƒ, eatApple: ƒ, eatCandy: ƒ}
```

Virtual Pet Player Object Solution 2

As with all challenges there are more than one solution. In this solution we incorporate lessons we have learned about Objects, Functions and Conditional Logic to build a Virtual Pet Player Object that meets our stated requirements in a slightly more real-world way (often referred to as **Don't Repeat Yourself**). Start by creating a property Method called **eat** with a function assigned that accepts a food parameter. Based on the value passed in (apple or candy) we can respond by adjusting the **health** and **fun** accordingly.

```
// Declare Object player with Method play
let player = {
  health: 100,
  fun: 0,
  play: function() {
```

```
    // increase fun by 10
    this.fun += 10;
  },
  eat: function(food) {
    if(food == 'apple') {
      // if apple,
        // increase health by 10
        this.health += 10;
    }
    else if(food == 'candy') {
      // if candy,
        // decrease health by 5
        this.health -= 5;
        // increase fun by 5
        this.fun += 5;
    }
  }
};

console.log('player object BEFORE:', player);
player.play();        // health is 100; fun is 10
player.eat('apple'); // health is 110; fun is 10
player.eat('candy'); // health is 105; fun is 15
console.log('player object AFTER:', player);
```

When we test we see the same results in the console for **health** and **fun**. Notice that we now have just two Methods display instead of three: **play** and **eat**.

Browser Developer Console output:

```
player object BEFORE:
?? {health: 100, fun: 0, play: ƒ, eat: ƒ}
player object AFTER:
?? {health: 105, fun: 15, play: ƒ, eat: ƒ}
```

ZENVA
Academy

While Loops allow us to execute code multiple times. We setup a **While Loop** block with a **Conditional Statement** that allows the code inside the loop to run repeatedly until the condition is no longer true. **While Loops** can be powerful in programming be great care must be taken in managing the condition. By this we mean the condition must have the ability to become **false** at some point otherwise we run the risk of creating an **Infinite Loop** which can cause your application and possibly your web server to crash!

Anatomy of a While Loop


```
// initialize total (if applicable)
let total = 0;
// initialize loop condition variable
let j = 1;

/* loop while this condition is TRUE;
condition is checked before each iteration of loop */
while(j <= 10) {
  // calculate total value
    total = total + j;
  // run iteration logic
    console.log('Iteration Sub-Total:', total);
```

```
  // loop condition variable incremented each iteration
    j++;
}

console.log('Final Total:', total);
```

Code Part	Explanation
let total = 0;	If applicable, initialize one or more variables that will be modified inside the loop.
let j = 1;	Initialize loop condition variable. This variable is defined and set with an initial value, often 0 or 1 based on your loop condition
while(j <= 10) {...}	This **While Loop Condition** can be any conditional statement. While this condition results in true; the code inside the loop is executed on every iteration. In this example as long as the value of j is less than or equal to **10** (j <= 10) the code inside the loop continues to execute. When this value is false the code after the loop executes; in this case the console.log()
total = total + j;	Sets total equal to total + j; (j is the current value of our loop condition variable). The value of total is updated on every iteration of the loop
console.log(total);	Prints updated total value to the console
j++;	Loop condition variable is incremented on each iteration of the loop. It is very important that this logic align with the condition to prevent an infinite loop. Notice that this increment occurs at the bottom of the loop. Everything inside the loop executes, this value is increased and then the **While Loop Condition** is checked again
console.log(total);	Prints final total value to the console. This statement follows the loop and is only executed once the **While Loop Condition** becomes false

Browser Developer Console output:

```
Iteration Sub-Total: 1
Iteration Sub-Total: 3
Iteration Sub-Total: 6
Iteration Sub-Total: 10
Iteration Sub-Total: 15
Iteration Sub-Total: 21
Iteration Sub-Total: 28
Iteration Sub-Total: 36
Iteration Sub-Total: 45
Iteration Sub-Total: 55
Final Total: 55
```

While Loop Example

The challenge is to create a Function, **sendSignal**, that we want to run **1000 times**. Each time the Function is executed it prints the message **HELP!** to the console. It is possible to simply create the function and call it a thousand times.

```javascript
// declare sendSignal Function
function sendSignal(){
  // print HELP! to the console
    console.log('HELP!');
}

// call sendSignal() Function - first time
sendSignal();
// call sendSignal() Function - second time
sendSignal();
// call sendSignal() Function - 1000 times
sendSignal();
```

Browser Developer Console output:

```
1000 HELP!
```

The **sendSignal** Function is called 1000 times and in the console we can see the number next to the text **HELP!** indicating that it printed a thousand times. This is not clean code however. In programming, if you are repeating yourself chances are there is a better way. This is a great use case for the **While Loop**.

```javascript
// declare sendSignal Function
function sendSignal(){
  // print HELP! to the console
    console.log('HELP!');
}

// initialize loop condition variable
let i = 0;

/* check if we have run loop 1000 times */
while(i < 1000){
  // call sendSignal() Function on each iteration
    sendSignal();
  // increment loop condition variable on each iteration
    //i = i + 1; (add 1 to i variable)
    //i += 1; (equivalent to above)
    i++; // (equivalent to above)
}
```

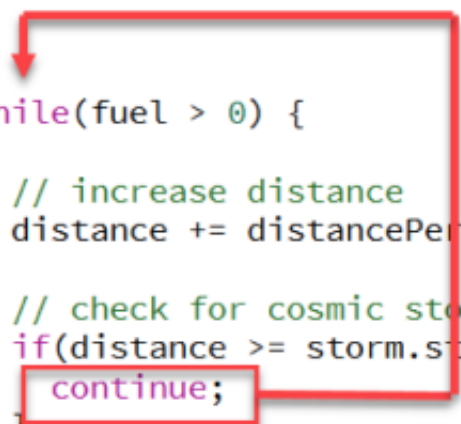**Browser Developer Console output**:

```
1000 HELP!
```

Notice we see the same output in the browser developer console but our method of execution is much cleaner.

In the last lesson we learned about **While Loops**. In this lesson we learn about two JavaScript keywords (**Continue** and **Break**) that can be used to control the flow of code inside a loop. Both of these keywords are generally used inside a **loop** and together with **Conditional Statements** and **Boolean Logic**.

Continue

Allows you to return to the **While Loop Condition** instead of continuing execution inside the loop if a given condition is met.

```javascript
while(fuel > 0) {

  // increase distance
  distance += distancePerFuel;

  // check for cosmic storm
  if(distance >= storm.start && distance <= storm.end) {
    continue;
  }

  // burn fuel
  fuel--;

  // stop if we arrive
  if(distance == planetDistance) {
    break;
  }
}
```

Break

Allows you to terminate execution of a loop if a given condition is met. This overrides the main **While Loop Condition**.

```javascript
while(fuel > 0) {

    // burn fuel
    fuel--;

    // stop if we arrive
    if(distance == planetDistance) {
        break;
    }
}
```
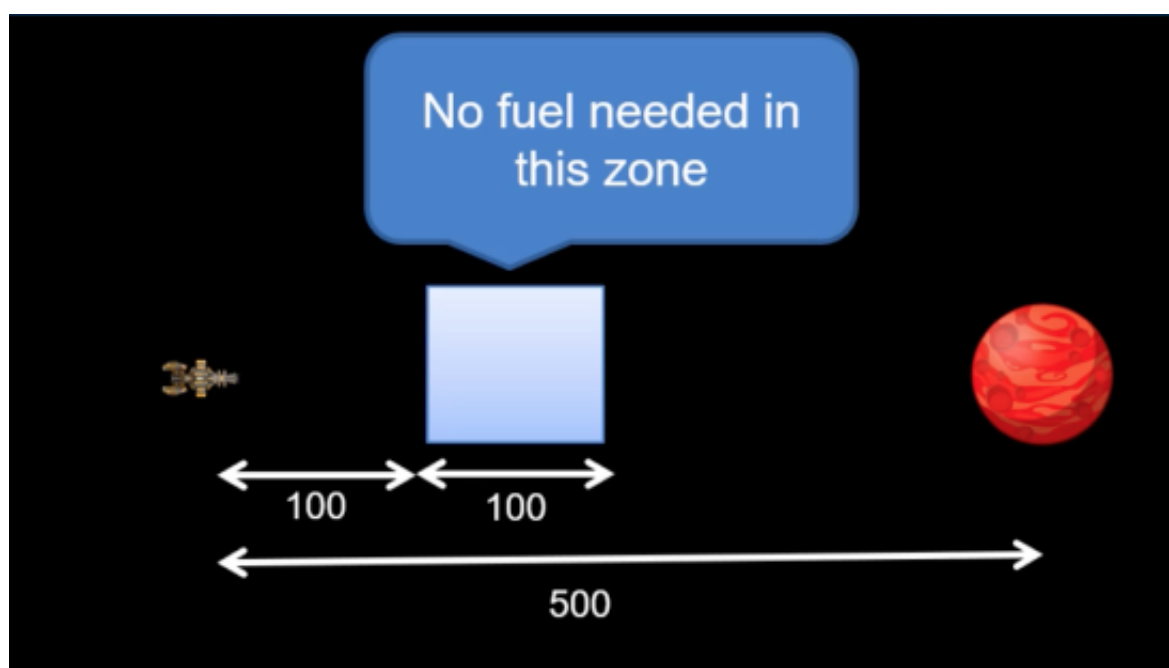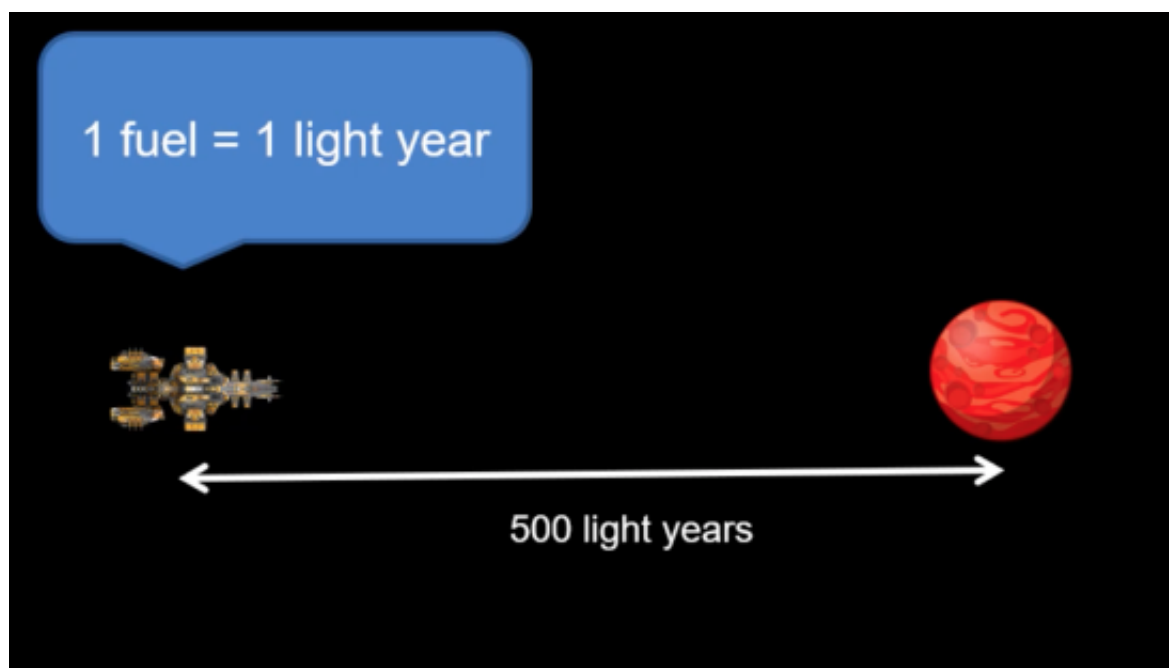
Terminate loop

Example

We can use **Continue** and **Break** inside a **While Loop** together with other concepts we have learned including **Conditional Statements** and **Boolean Logic** to solve this lesson's challenge.

What we need to determine

Can we travel a distance of 500 light years with the fuel we have available?

What we know

The distance we need to travel is 500 light years
We have 1000 units of fuel
We move without consuming fuel when certain conditions are true. These conditions occur between a distance of 100 light years and 200 light years
We move while consuming fuel in all other areas of the journey. It takes 1 fuel unit to travel a distance of 1 light year
Once 500 light years have been travelled; all further travel stops

Since the action of consuming fuel and moving is repeated multiple times; a **While Loop** provides a great solution.

```
let fuel = 1000;
let distance = 0;

// setup while loop with condition
while(fuel > 0) {

    distance++;

    if(distance >= 100 && distance < 200) {
        continue;
    }

    fuel--;

    if(distance == 500) {
        break;
    }
}

console.log('Fuel Remaining:', fuel);
console.log('Distance Travelled:', distance);
```

Code Part	Explanation
let fuel = 1000;	Defines a variable fuel and sets it equal to the amount we currently have (1000 fuel units). The value of fuel is now 1000
let distance = 0;	Defines a variable distance that will represent how far we have travelled. Since we have not moved yet we set its initial value to be zero. The value of distance is now 0
while(fuel > 0) {	Based on the assumption that the ship will keep moving as long as it has fuel we setup our **While Loop** with a **Condition** of fuel must be greater than zero (fuel > 0)
distance++;	As long as fuel > 0 ; **increment** distance by 1 each iteration
if(distance >= 100 && distance < 200){continue;}	Check for known distance condition that states we move without consuming fuel when distance >= 100 AND distance < 200. If **TRUE** we use the **continue** keyword to return to **While Loop Condition** without executing remainder of while loop code block
fuel--;	If previous distance check evaluates to **FALSE**; **decrement** fuel by 1 each iteration
if(distance == 500){break;}	Check for known distance condition that tells us if we have reached our destination. If **TRUE** we use the **break** keyword to exit **While Loop** without checking **While Loop Condition**
console.log('Fuel Remaining:', fuel);	Prints Fuel Remaining: 600 in the console
console.log('Distance Travelled:', distance);	Prints Distance Travelled: 500 in the console

Browser Developer Console output:

```
Fuel Remaining: 600
Distance Travelled: 500
```

If we have fuel remaining and the distance travelled is 500 light years our mission has been successful.

In the last two lessons we learned about **While Loops** and two JavaScript keywords **continue** and **break** to control flow within the loop. In this lesson we look at an alternative loop **For Loops**. We setup and use this loop differently but it's important to note it's still a loop and we can still use the same JavaScript keywords (**continue** and **break**) to control flow within it.

Comparison of For Loops and While Loops

The **For Loop** (keyword: for) and **While Loop** (keyword: while) have a common use and common setup but differs in syntax. The **Initialization Condition Index**, **Conditional Statement**, and **Modification of Condition Index** occur in different locations.

Code Part	For Loop Location	While Loop Location
Initialize Condition Index	Inside for(...) statement	Outside while(...) statement
Conditional Statement	Inside for(...) statement	Inside while(...) statement
Modify Condition Index	Inside for(...) statement	Inside while body {...}
Code Execution Body	Inside for body {...}	Inside while body {...}

Anatomy of a For Loop

The **for statement** creates a loop that consists of three optional expressions, enclosed in parentheses and separated by semicolons, followed by a code block enclosed in curly braces {statements}. It has the following **syntax** for([initialization]; [condition]; [final-expression]) {statements}

```
for(let i = 0; i < 10; i++) {
    console.log('Iteration:', i);
}
```

Sequence	Code Part	Explanation
1. initialization	let i = 0;	Variable declaration evaluated once before the loop begins. Typically used to initialize a counter variable
2. condition	i < 10;	An expression to be evaluated before each loop iteration. If this expression evaluates to true, statements are executed
3. statements	console.log(i);	Code statements that are executed as long as the condition evaluates to true
4. final-expression	i++;	An expression evaluated after each loop iteration and before next evaluation of condition. Generally used to update or increment the counter variable

Browser Developer Console output:

```
Iteration: 1
Iteration: 2
Iteration: 3
```

```
Iteration: 4
Iteration: 5
Iteration: 6
Iteration: 7
Iteration: 8
Iteration: 9
```

For Loop Example

This lesson's challenge is to simulate population growth in a city over a ten year period. We know the **current population** is **100** and that the expected growth rate is **5% per year**. What will be the population in 10 years?

```
// initialize population variable
let population = 100;

// setup for-loop to iterate 10 times (10 years)
for(let i = 0; i < 10; i++) {
  // increase population variable by 5% each iteration
  // same as: population = population * 1.05;
    population *= 1.05;
}

// print final population to console
console.log('Population after 10 years:', population);
```

Browser Developer Console output:

```
Population after 10 years: 162.8894626777442
```

Arrays are list-like objects, common to most programming languages, that are used to represent **ordered collections**. They are **collections** in that they can represent a list of any size containing any **Data Type**. They are considered **ordered** in that they maintain a sequential numerical **index** (represented as an integer) from **zero** to **array length – 1**.
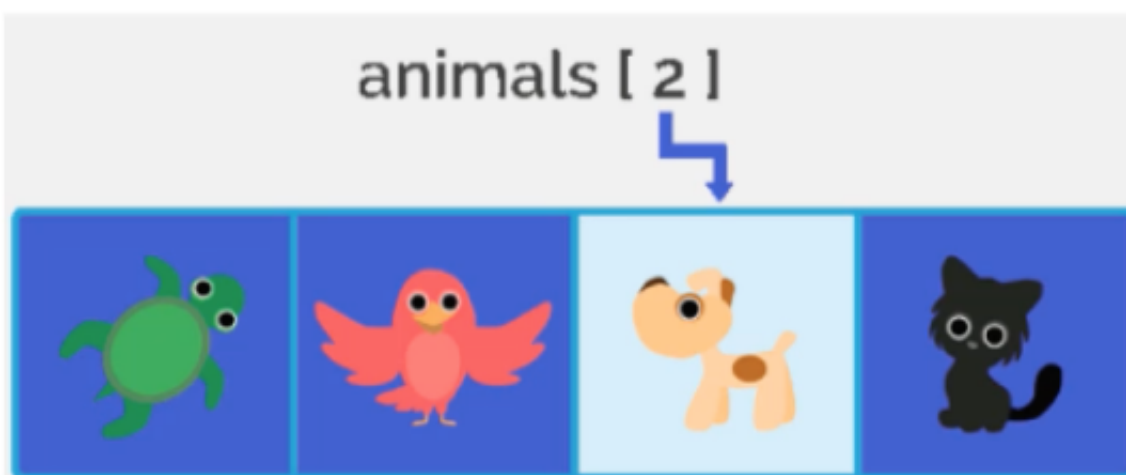
Create an Empty Array

To create an empty array, set a variable equal to empty square brackets. The output in the console shows **[]** representing an empty array along with our console log of the array length showing zero.

```
let animals = [];
console.log('Array:', animals);
console.log('Array Length:', animals.length);
```

Browser Developer Console output:

```
Array: ?? []
Array Length: 0
```

If you open the **array** Object in the console by clicking the arrow (▶) you will also notice a **length property** equal to zero and a __proto__ property which contains a list of methods built-in to arrays that make them an extremely powerful and flexible programming tool (we will only cover a few basic methods in this course).

```
Array: ? []
    length: 0
  ?? __proto__: Array(0)
Array Length: 0
```

Create an Array with Elements

You can create an an array and assign it initial elements at the same time. In this example we **add** four elements to an array and assign the array to a variable named **animals**.

```
let animals = ['turtle', 'cat', 'dog', 'bird'];
console.log('Array:', animals);
console.log('Array Length:', animals.length);
```

Browser Developer Console output:

```
Array: ?? (4) ["turtle", "cat", "dog", "bird"]
Array Length: 4
```

Expand the array (▶) and notice the four elements with indexes **0** to **3** with length of **4**.

```
Array: ? (4) ["turtle", "cat", "dog", "bird"]
        0: "turtle"
        1: "cat"
        2: "dog"
        3: "bird"
        length: 4
  ??    __proto__: Array(0)
Array Length: 4
```

Access First Array Element

Array indexes start at position **0**. So to access the **first** element of the **animals array** we reference **animals[0]**

```
// access first array element
let animals = ['turtle', 'cat', 'dog', 'bird'];
let animal = animals[0];
console.log('Array:', animals);
console.log('Animal at index 1:', animal);
```

Browser Developer Console output:

```
Array: ?? (4) ["turtle", "cat", "dog", "bird"]
Animal at index 0: turtle
```

Access Last Array Element

Since array indexes start at position **0**, the **last** element of the **animals array** is at **animals[animals.length - 1]**
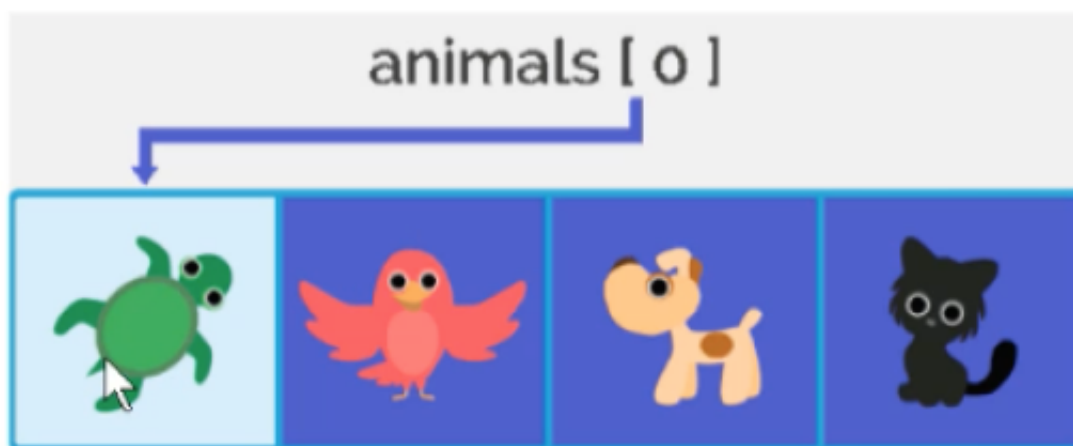
```
// access last array element
let animals = ['turtle', 'cat', 'dog', 'bird'];
let animal = animals[animals.length – 1];
console.log('Array:', animals);
console.log('Last Animal in array:', animal);
```
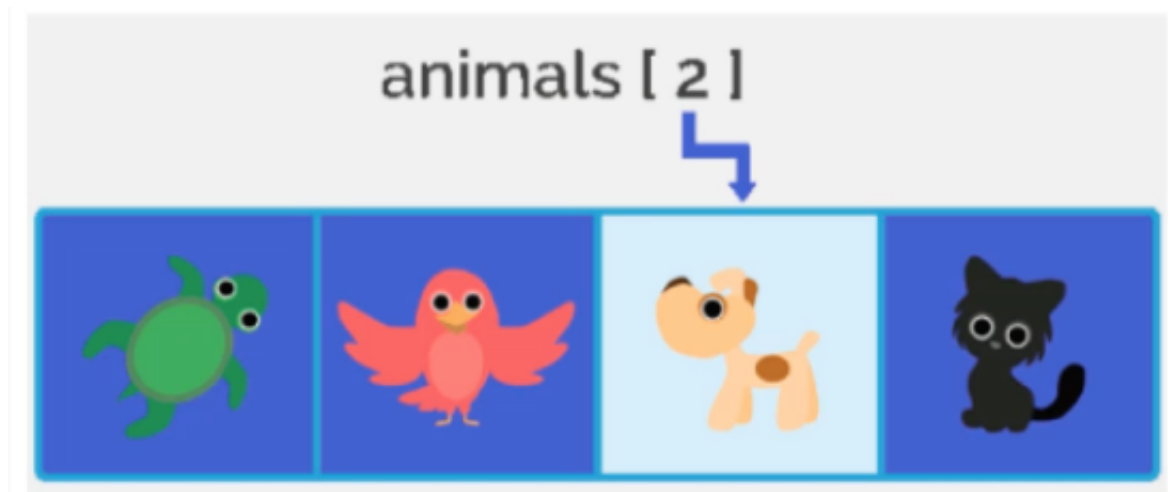
Browser Developer Console output:

```
Array: ?? (4) ["turtle", "cat", "dog", "bird"]
Last Animal in array: bird
```

Access Any Array Element

If you want to access a specific element (**n**) in an array simply subtract one from the element number you want (**n - 1**) and that is your **index**. Access the third element:

```
let animals = ['turtle', 'cat', 'dog', 'bird'];
let animal = animals[2];
console.log('Array:', animals);
console.log('Animal at index 2:', animal);
```

Browser Developer Console output:

```
Array: ?? (4) ["turtle", "cat", "dog", "bird"]
Animal at index 2: dog
```

Changing Array Elements

Change the first element in the array from **turtle** to **dinosaur**:

```
let animals = ['turtle', 'cat', 'dog', 'bird'];
animals[0] = 'dinosaur';
console.log('Array:', animals);
console.log('Animal at index 0:', animals[0]);
```

Browser Developer Console output:

```
Array: ?? (4) ["dinosaur", "cat", "dog", "bird"]
Animal at index 0: dinosaur
```

Add Element to Array

JavaScript Arrays have a lot of built-in methods. To add an element to the end of an array we can use the **push** method. Notice that we now have **5** elements in the array with the last one being the **lizard** we just added.

```
let animals = ['turtle', 'cat', 'dog', 'bird'];
```

```
animals.push('lizard');
console.log('Array:', animals);
```

Browser Developer Console output:

```
Array: ?? (5) ["turtle", "cat", "dog", "bird", "lizard"]
```

Remove Element from Array

JavaScript Arrays have a lot of built-in methods. To remove an element from the end of an array we can use the **pop** method. This method takes no parameters. Notice that we now have **4** elements in the array because we removed the last one.

```
let animals = ['turtle', 'cat', 'dog', 'bird', 'lizard'];
console.log('Array BEFORE:', animals);
animals.pop();
console.log('Array AFTER:', animals);
```

Browser Developer Console output:

```
Array BEFORE: ?? (5) ["turtle", "cat", "dog", "bird", "lizard"]
Array BEFORE: ?? (4) ["turtle", "cat", "dog", "bird"]
```

This has been a basic introduction to **arrays**. In the next two lessons we will dive deeper into additional array features.

In the last lesson we learned the basics of **Arrays**. One of the most powerful aspects of these **ordered collections** is the ability to easily iterate over them and perform actions, calculations and more for each Array element. There are many ways to iterate over Arrays and in this lesson we will cover three: **While Loop**, **For Loop**, and a built-in JavaScript Array method called **forEach**. It's important to keep in mind that the first element of an Array is at index **0** and the last element is at index **length - 1**. Let's say we have an Array of **scores** and we need to increase all scores by **1**

While Loop

One way to accomplish this is to iterate through the **scores** Array using a **While Loop** and increase one score element on each iteration of the loop until all scores have been increased.

```
let scores = [10, 20, 10];

console.log('Scores Array BEFORE:', scores);
// using a while loop increase scores by 1
let i = 0;
while(i < scores.length) {
  scores[i]++;
  i++;
}
console.log('Scores Array AFTER:', scores);
```

Browser Developer Console output:

```
Scores Array BEFORE: ?? (3) [10, 20, 10]
Scores Array AFTER: ?? (3) [11, 21, 11]
```

Code Step	Explanation
let scores = [10, 20, 10];	Declare and initialize an Array of three values assigning the Array to the variable scores
let i = 0;	Initialize loop condition variable. This variable is defined and set with an initial value of 0 since Arrays are zero-based
while(i < scores.length) {...}	This **While Loop Condition** checks if our loop condition variable is less than the length of the Array. While this condition results in true; the code inside the loop is executed on every iteration
scores[i]++;	Sets the score value of each element equal to score + 1; (score is the current element of the Array). In this example, we iterate through the loop three times since the **length of the Array is 3**. First time scores[0] = scores[0] + 1 which is 10 + 1 so the final result of first Array element changes from 10 to 11. The same process happens the second and third iteration of the loop
i++;	Loop condition variable is incremented by 1. Everything inside the loop executes, this value is increased and then the **While Loop**

Condition is checked again

For Loop

Another way to accomplish this is to iterate through the **scores** Array using a **For Loop** and increase one score element on each iteration of the loop until all scores have been increased. This is simply a different approach than above. Notice that the output in the developer console is exactly the same.

```
let scores = [10, 20, 10];

console.log('Scores Array BEFORE:', scores);
// using a for loop
for(i = 0; i < scores.length; i++){
  scores[i]++;
}
console.log('Scores Array AFTER:', scores)
```

Browser Developer Console output:

```
Scores Array BEFORE: ?? (3) [10, 20, 10]
Scores Array AFTER: ?? (3) [11, 21, 11]
```

Code Step	Explanation
let scores = [10, 20, 10];	Declare and initialize an Array of three values assigning the Array to the variable scores
for(initialization; condition; final-expression) {statements}	Setup **for statement** with its three expressions, enclosed in parentheses and separated by semicolons, followed by a code block enclosed in curly braces {statements}
let i = 0;	**Initialization**. Initialize loop condition variable. This variable is defined and set with an initial value of 0 since Arrays are zero-based
i < scores.length;	**Condition**. Checks if our loop condition variable is less than the length of the Array
scores[i]++;	**Statements**. Sets the score value of each element equal to score + 1; (score is the current element of the Array). In this example, we iterate through the loop three times since the **length of the Array is 3**. First time scores[0] = scores[0] + 1 which is 10 + 1 so the final result of first Array element changes from 10 to 11. The same process happens during the second and third iteration of the loop
i++;	**Final-expression**. Loop condition variable is incremented by 1 after all statements inside the loop have been executed. After this value is modified the **For Loop Condition** is checked again

forEach Array Method

The **forEach** built-in JavaScript Array method executes a provided function once for each Array element. The provided function is known as a **callback** and can take three parameters:

Parameter	Explanation
currentValue	**Required**. The current element being processed in the Array; in this case we name the parameter entry. It is **important** to note that any Number, String, or Boolean passed to a function is **passed by value** which means it is a **copy** of the original value. This is important because if you modify this value inside the function, the original value outside the function will not change. Therefore, the original Array together with its corresponding index can be used inside the function to modify the original Array (if required)
index	**Optional**. The index of currentValue in the Array
Array	**Optional**. The Array forEach() was called upon; in this case the Array scores. It is sometimes necessary to modify the original Array inside a function. Since Arrays and Objects are **passed by reference** to functions (the original; not a copy); the original Array together with its corresponding index can be used inside the function to modify the original Array element

```javascript
let scores = [10, 20, 10];

console.log('Scores Array BEFORE:', scores);
// using forEach JavaScript Array method
scores.forEach(function (entry, index, scores) {
    scores[index]++;
});
console.log('Scores Array AFTER:', scores);
```

Browser Developer Console output:

```
Scores Array BEFORE: ?? (3) [10, 20, 10]
Scores Array AFTER: ?? (3) [11, 21, 11]
```

Challenge

In this lesson's challenge we need go through each entry in a course catalog and if the author is **Zenva**, add an additional copy of the course. The catalog is an Array of Objects with properties: title, author, and copies. To solve this challenge we will iterate through the catalog using the JavaScript Array **forEach** method. For each catalog entry we use an **IF Statement** to determine if the author is Zenva. If so, we increase the number of copies in our catalog. Notice that we can modify entry.copies inside the function since **entry** represents an element of the **catalog Array** that is an **Object** and Objects are **passed by reference** to functions (the original; not a copy) and

can therefore be modified directly.

```javascript
let catalog = [
    {
        title: 'JS for Beginners',
        author: 'Zenva',
        copies: 1
    }, {
        title: 'HTML for Beginners',
        author: 'Zenva',
        copies: 1
    }, {
        title: 'CSS for Beginners',
        author: 'XYZ',
        copies: 1
    }
];

console.log('Catalog BEFORE:', catalog);

catalog.forEach(function (entry) {
    if (entry.author == 'Zenva') {
        entry.copies++;
    }
});
console.log('Catalog AFTER:', catalog);
```

Browser Developer Console output:

```
Catalog BEFORE:
? (3) [{…}, {…}, {…}]
  ?? 0: {title: "JS for Beginners", author: "Zenva", copies: 1}
  ?? 1: {title: "HTML for Beginners", author: "Zenva", copies: 1}
  ?? 2: {title: "CSS for Beginners", author: "XYZ", copies: 1}
Catalog AFTER:
? (3) [{…}, {…}, {…}]
  ?? 0: {title: "JS for Beginners", author: "Zenva", copies: 2}
  ?? 1: {title: "HTML for Beginners", author: "Zenva", copies: 2}
  ?? 2: {title: "CSS for Beginners", author: "XYZ", copies: 1}
```

Notice that in the **Catalog AFTER** the number of copies where **Zenva** is the author has increased while the other course has remained the same.

Multidimensional Arrays are Arrays that contain other Arrays. For example: let mArr = [[1,2,3],[4,5,6,7]]. This allows for an intuitive way to store 2D, 3D, or even higher dimensional values. In the simplest case, for example a 2D Array, you can think of it like a table with rows, columns, and an intersecting cell containing information like in a spreadsheet.

Declaring Multidimensional Arrays

Considering the multidimensional Array shown above we can visualize 4 rows and 4 columns. Let's start by coding a simple example.

```
let a = [[1, 2, 3], [4, 5, 6, 7]];
console.log(a);
```

Browser Developer Console output:

```
?? (2) [Array(3), Array(4)]
```

We can see in the console an Array that contains two elements both of which are Arrays; a multidimensional Array. We can also notice that the first Array has three elements and the second has four elements. Let's open it up further in the console to see inside the multidimensional Array. The first Array is at index zero and the second Array is at index one. Also notice the **length property** of the main Array.

```
? (2) [Array(3), Array(4)]
  ?? 0: (3) [1, 2, 3]
  ?? 1: (4) [4, 5, 6, 7]
    length: 2
```

Accessing Array Elements

To help visualize how to access Array elements, you can open up the Arrays inside the main Array to see each index and its value.

```
? (2) [Array(3), Array(4)]
  ? 0: Array(3)
       0: 1
       1: 2
       2: 3
       length: 3
     ?? __proto__: Array(0)
  ? 1: Array(4)
       0: 4
       1: 5
       2: 6
       3: 7
       length: 4
     ?? __proto__: Array(0)
    length: 2
  ?? __proto__: Array(0)
```

Let's write code to log to the console the value of **6** from the second Array. Since Arrays are zero-based, use its index position to return the corresponding value. Access the main Array first by typing **a** inside the console.log() statement. Next in the first set of square brackets identify which element of the main Array contains the element we are looking for; in this case **1** since it's the second element. Finally, add a second pair of square brackets and use the index associated with the value we are seeking; in this case it's the third element (n) which is index **2** (n - 1)

```
let a = [[1, 2, 3], [4, 5, 6, 7]];

console.log(a[1][2]);  // 6
```

Browser Developer Console output:

```
6
```

Modifying Array Elements

The same Array index technique can be used to change an array element. Change the element we just selected from the value **6** to the value **100** using the Array selection and Assignment Operator (**=**)

```
let a = [[1, 2, 3], [4, 5, 6, 7]];

console.log('BEFORE:', a[1][2]);  // 6

a[1][2] = 100;

console.log('AFTER:', a[1][2]);  // 100
```

Browser Developer Console output:

```
BEFORE: 6
AFTER: 100
```

Challenge

In this lesson's challenge we need to map an area of made up of three types of terrain: desert, grass, and water. Our assignment is to divide the map into rows and columns showing the location of each type of terrain. For this we will use a multi-dimensional array that is essentially a table where each row and column combination creates a cell where the type of terrain in that area can be stored. This would look similar to creating a table in a spreadsheet.
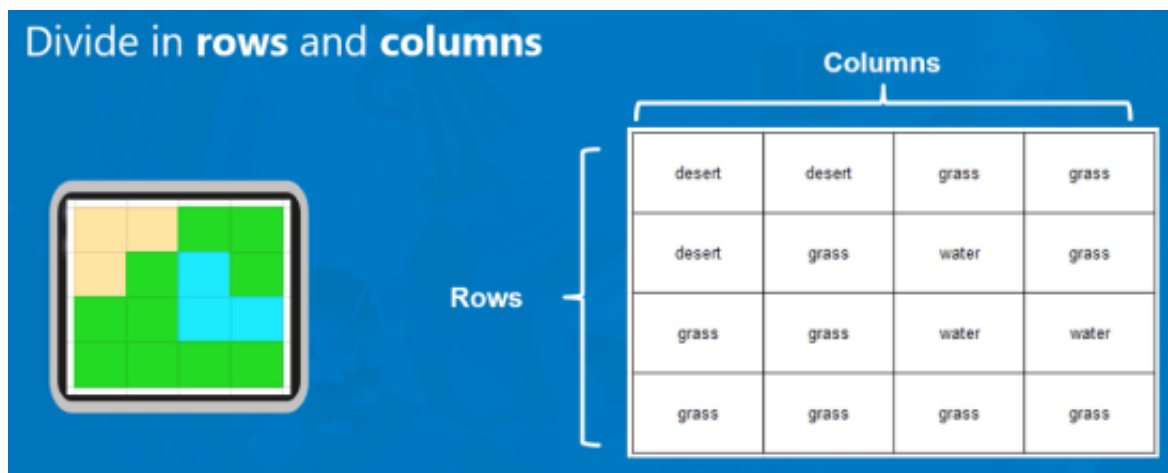
Let's start by declaring a variable named **terrains** and assign it to an Array that contains four other Arrays, like rows in a table, matching the terrain in the image above.

```
let terrains = [
  ['desert', 'desert', 'grass', 'grass'],
  ['desert', 'grass', 'water', 'grass'],
  ['grass', 'grass', 'water', 'water'],
  ['grass', 'grass', 'grass', 'grass']
];

console.log(terrains);
```

Browser Developer Console output:

```
? (4) [Array(4), Array(4), Array(4), Array(4)]
  ?? 0: (4) ["desert", "desert", "grass", "grass"]
  ?? 1: (4) ["desert", "grass", "water", "grass"]
  ?? 2: (4) ["grass", "grass", "water", "water"]
  ?? 3: (4) ["grass", "grass", "grass", "grass"]
    length: 4
  ?? __proto__: Array(0)
```

Even though this is a larger Array, we can access any Array element just as we did in the section above. Use the following diagrams to help visualize the process of accessing the value **desert** from the second row (**index 1**); first column (**index 0**)

```
let terrains = [
  ['desert', 'desert', 'grass', 'grass'],
  ['desert', 'grass', 'water', 'grass'],
  ['grass', 'grass', 'water', 'water'],
  ['grass', 'grass', 'grass', 'grass']
];

let el = terrains[1][0];

console.log(el); // desert
```
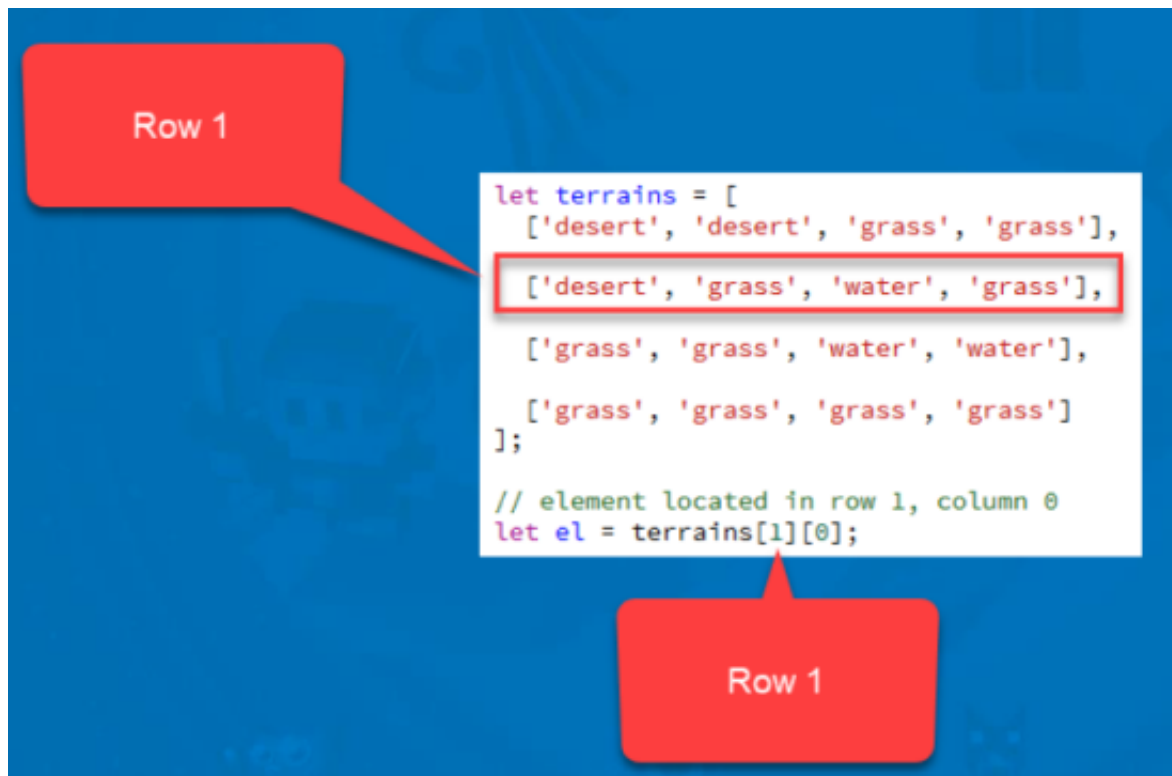
https://academy.zenva.com

Browser Developer Console output:

```
desert
```

The **DOM** refers to the **Document Object Model** which is the JavaScript representation of the **HTML** on a web page. The HTML page is made up of elements which are created using **tags** such as <body>, <h1>, and <p> as shown below:

```
<body>
   <h1 id="first-heading">JavaScript</h1>
   <p>Learn coding at <a href="https://zenva.com/">Zenva</a></p>
</body>
```

The DOM is represented in the browser by a tree-like structure where you start with a top node and then you have sub-nodes that contain all of the content.

The **DOM API** is the **Application Programming Interface** that JavaScript uses to access and modify DOM elements. In this lesson we'll learn DOM API basics such as selecting an element by its ID, changing its text or HTML content, and lastly modifying its CSS style properties.

Challenge

Our client's website welcome sign has been hacked. Its message has been changed along with its color. Since the sign uses HTML, our challenge is to access the DOM from JavaScript to **select** the DOM element we need to change, **modify** its **text** or **innerHTML**, and finally to change its **color** using the DOM element's **style** property. Specifically we need to:

Requirements

Change welcome sign text back to "Welcome, travelers!"
Change the font color back to **blue**

Create DOM Elements

The first thing to do is re-create the welcome sign from our client's web page to we can work on a solution from JavaScript to meet the requirements we defined above. To do this we add a HTML <div> tag element and give it an ID, style to set its color, and include the hacked welcome text to be displayed on the web page.

```
<!-- index.html -->
<div id="sign" style="color:red;">Welcome, pirates!</div>

<script src="script.js"></script>
```

Browser Web Page:

Select DOM Element

We now switch our focus to the JavaScript file. When working with the DOM, it's important to remember that if we want to affect it we must select it. We select a DOM element from JavaScript by using the **document** Object. One method available to us is **getElementById** (be cautious of the method's casing... it needs to be exact). This method allows us to select a specific DOM element based on **ID**. The ID we provided above is **sign** so that is what we use to select the associated DOM element. Notice in the Browser Developer Console that the <div> with an ID of **sign** has been selected.

```
// script.js
// 1. Select
let sign = document.getElementById('sign');
console.log(sign);
```

Browser Developer Console output:

```
<div id="sign" style="color:red;">Welcome, pirates!</div>
```

Modify Text Content

If you need to modify the actual HTML content you could use the **document.innerHTML** method. If you just need to modify the HTML element's Text content you could use the **document.textContent** method.

```
// script.js
// 1. Select
let sign = document.getElementById('sign');

// 2. Modify
```

```
console.log('BEFORE:', sign.textContent);

sign.textContent = 'Welcome, travelers!';

console.log('AFTER:', sign.textContent);
```

UPDATED **Browser Web Page**:

Welcome, travelers!

Browser Developer Console output:

```
BEFORE: Welcome, pirates!
AFTER: Welcome, travelers!
```

Change DOM Element Style

We now have the correct message showing on the client's web page but it still does not look quite right. Our next step is to change the HTML element's font color. We do this by accessing the HTML element's style property which gives us access to all of the CSS for the DOM element so you can change color, size, and more. In fact, you can access all CSS properties and the assignment to the element's **style** property is any value that is valid CSS.

```
// script.js
// 1. Select
let sign = document.getElementById('sign');

// 2. Modify
sign.textContent = 'Welcome, travelers!';

// 3. change style
sign.style.color = 'blue';
```

UPDATED **Browser Web Page**:

Welcome, travelers!

Congratulations! Mission Complete!