anonymous function

A function without a name is referred to as an **anonymous function**. These occur when the function code is written in-line or set equal to an event, as opposed to as a separate location that must be referenced by name. Example of anonymous function:

```
myButton.onclick = function() {
   alert('You clicked the button!');
};

// or:
myButton.addEventListener('click', function() {
   alert('You clicked the button!');
});
```

Also see **function**

---

argument / parameter

- Data that goes between the parentheses of a method or a function is called an **argument**.
- When the function is defined, the data in the parentheses, known as **parameter(s)** is defined as a variable.
- When the function is called, an argument is passed into the parentheses. In this way, the argument sets the value of the parameter:

```
function greetUser(name) {
   console.log(`Hello, ${name}!`);
}
greetUser("Brian"); // Hello, Brian!
```

---

array

An array is a type of **object** that store data as numbered list items. The syntax is to put the values, which can be of any datatype, inside of square brackets. Items are referenced by their numeric position, known as the **index**, which start numbering at 0.

```
let fruits = ['kiwi', 'pear', 'plum'];
console.log(fruits[0]); // kiwi
```

Also see **index**, **object**.

---

arrow function

An arrow function uses **=>** instead of the keyword **function**. An arrow function implicitly returns a value, so for functions with only one line of logic, the **return** keyword and curly braces **{}** may be omitted. This concise syntax makes arrow

functions the preferred way of writing callbacks, which are anonymous functions passed to other functions. Here we declare an array of numbers followed by calling the sort() method on the array to put the items in ascending order. The sort() method takes a callback, for which we are using arrow syntax:

```
const nums = [13, 6, 45, 2, 89, 11, 4, 23];
nums.sort((a,b) => a - b);
console.log(nums); // [2, 4, 6, 11, 13, 23, 45, 89]
```

---

assignment operator

The equals sign **=** is known as the **assignment operator**, because it assigns or sets a value: **x = 5** assigns **5** as the value of **x**.

```
let x = 8;
x = 9;
```

---

asynchronous

The term **asynchronous** (async, for short) refers to code being executed at different times, particularly due to delays in data being made available as in a server request. One function makes a request for data from a server, and another function must asynchronously wait for that data to load and be made available. Callback functions are used to handle asynchronous requests.

---

block scope

Block refers to variables that are available only within the particular code block in which they are declared, as defined by a pair of curly braces { }. Also see **scope**, **function scope**, and **global scope**.
The rules for **variable scope**:

- variables not declared inside any { } are **global**, which means they are available everywhere in the script.
- **let** and **const** declared inside { } are **block scoped**, which means they are available only inside that code block.
- **var** declared inside { } are **global** (unless the { } is a function)
- Any variable declared inside a function is available only to that function. These are known as **local variables** or **function-scoped variables**.

---

boolean

A variable or value that can be only **true** or **false**. Conditional logic resolves to a boolean:

```
let isOnline = true;
isOnline = false;
```

block (of code)

A code block is all the code that occurs between a set of curly braces. When declared with let or const, any variable declared inside a block is available only to that block and is therefore called **block scoped**. The main types of blocks are functions, if-statements and loops:

```
// code block / block scope examples
// 1. function
function greetUser(username) {
   return `Welcome ${username}`;
}

// 2. for loop
// i is scoped to the block (does not exist outside the block)
function greetUser(username) {
for(let i = 0; i < 5; i++) {
   console.log(i); // 0, 1, 2, 3, 4
}
console.log(i); // undefined (i only exists inside its block)

// 3. if statement
let x = 5;
// if x is greater than 2, square it and save the value to y
if(x > 3) {
   let y = x ** 2;
   console.log(y); // 25
}
console.log(y); // undefined (y only exists inside its block)
```

callback

An anonymous function that is passed to another function as its argument is known as a **callback**. Callbacks frequently have arguments of their own. Here we declare an array of numbers followed by calling the sort() method on the array to put the items in ascending order. The sort() method takes a callback:

```
const nums = [13, 6, 45, 2, 89, 11, 4, 23];
nums.sort(function(a,b) {
   return a - b;
});
console.log(nums); // [2, 4, 6, 11, 13, 23, 45, 89]
```

comment

Comments are notes or commented out code which are ignored by the browser.

```
// declare a variable
let fruit = "apple";
// alert(fruit);
```

---

condition

A condition is something that is evaluated and resolves to *true* or *false*. Conditions are found in loops and in if-else statements (conditional logic). The condition of a loop must be true for the loop's code block to execute. In this example, the condition is **x < 10**. If the condition is true, the code inside the {} will run:

```
let x = 9;
if(x < 10) {
   x++;
}
```

---

console

The **console** is a browser debugging environment where you can output code via the command. The following would produce output of **25** in the console, which may be accessed via Dev Tools or **Inspect > Console**: `let x = 5; console.log(x * x); // 25`.

---

const

The **const** keyword for declaring primitive variables (string, number, boolean) prevents them from being changed. Objects declared with **const** can have their properties and items changed by cannot have their data type changed.

---

counter variable

A variable that exists to keep count of something. Loops rely on counter variables to be incremented or decremented with each iteration of the loop so that the condition driving the loop can ultimately become false and the loop can therefore stop.

---

document

The webpage, a child object of the Window. JS "sees" the document and all of its elements as a hierarchy of so many nested objects, which JS can manipulate in a myriad of ways.

DOM (Document Object Model)

JavaScript "sees" the entire web page as a hierarchical tree of objects, known as the DOM, with the Window being the top-level object. The structure is analogous to how in HTML, we have a nested hierarchy of tags.

dot syntax

The use of dots to access or “drill down into” objects is known as dot syntax. In this example, **triStates.ny.capitol** returns the value of the **capitol** property, "Albany", which is a child of the **ny** property, itself an object, which is a child of the **triStates** object:

```
const triStates = {
  ct: {
    nickname: "Constitutino State",
    capitol: "Hartford"
  },
  nj: {
    nickname: "Garden State",
    capitol: "Trenton"
  },
  ny: {
    motnicknameto: "Empire State",
    capitol: "Albany"
  }
}

console.log(triStates.ny.capitol); //  Albany
console.log(triStates.ny); // {nickname: "Empire State", capitol: "Albany"}
console.log(triStates.nj.motto); //  Garden State
console.log(triStates.nj); // {nickname: "Garden State", capitol: "Trenton"}
```

else

Conditional logic is often expressed by if-else statements: if a condition is true, do something; **else** do something else:

```
let x = 2;
if (x > 3) {
  console.log('The condition is true!');
} else {
  console.log('The condition is false!');
}
```

Also see **if-else logic**.

event

An **event** is something that happens during the running of an application. Events are often used as triggers to call functions. Common events include **click** and **change**:

```
myButton.addEventListener('click', doSomething);
mySelectMenu.addEventListener('change', doSomethingElse);
```

---

for loop

A for loop executes a block of code again and again, as long as a condition is true. The condition must become false eventually, or else the loop never ends (infinite loop). A for loop considers three pieces of information between its parentheses in order to decide if it will run the code inside its curly braces: a **counter**, a **condition** and an **incrementer**:

```
for(let i = 0; i < 10; i++) {
  console.log(i*i); // 1, 4, 9, 16, 25, 36, 49, 64, 81, 100
}
```

---

function

A function is a block of code that runs only when it is invoked (called). The function can be called in the code or by an event on the web page, such as a button click. A function usually (but not always) has a name to call it by:

```
function greetUser(name) {
  console.log(`Hello, ${name}!`);
}

greetUser("Brian"); // Hello, Brian!
myButton.addEventListener('click', greetUser);
```

---

function scope / local scope

Function scope / local scope refers to variables that are available only inside the function in which they are defined. Also see **scope**, **global scope** and **local scope**.

---

global/ window object

The highest level object in JS is the **global window object**, the direct children of which include the *document*, **console**, **navigation** and **history** objects. The window object is assumed and need not be referenced:

```
alert("Hello!");
// or:
window.alert("Hello again!");
```

---

global scope, global memory

Global scope / memory refers to variables that are declared at the level of the **global object** and as such are available throughout a script, as opposed to being confined to the scope of a particular code block. Also see **scope**, **function scope**

and **local scope**. In this example **x** is global because it is declared outside of any code block, but **i** is scoped to the for loop, and so is *not* available outside the code block. Attempts to reference it result in **ERROR not defined**.

```
let x = 5; // global variable

for(let i = 0; i < 10; i++) {
    console.log(i);
}

console.log(x); // 5
console.log(i); // ERROR i is not defined
```

---

history object

A direct child of the window, the **history object** has methods pertaining to the browser history. Although the *window* part is optional, it is customary to include window when referencing the history object:

```
    // reload the last visited browser page
    window.history.back();
```

---

hoisting

A function can be called before it is defined in the code, because functions are automatically hoisted (lifted) to the top of the code block. By contrast, variables are not hoisted and therefore can only be referenced after they have been declared.

```
// calling the function above where it is defined works, because functions are hoisted (lifted) to the top of the code
sayHey("Joe"); // Hey, Joe!

function sayHey(name) {
    console.log(`Hey, ${name}!`);
}

// referenceing a variable above where it is declared throws an erro, because variables are NOT hoisted (lifted) to the top of the code
console.log('Bunny likes ' + veg); // Error: veg is not defined
let veg = 'kale';
```

---

if-else logic, if-statement

The logic of decision making in programming, conditional logic is often expressed by if-else statements: if a condition is true, do something; else do something else:

```
let x = 10;
let y = 3;

if(x > y) {
   console.log('x is greater than y!');
} else {
   console.log('x is equal to or less than y!');
}
```

Also see **conditional logic**.

---

increment

To increment means to increase or go up. To increment a variable means to have its numeric value go up. In loops, a counter variable is incremented or decremented each time through the loop.

```
// square the counter variable i each time the loop runs
// each time the loop runs, increment i by i with i++
for(let i = 0; i <= 5; i++) {
   console.log(i ** 2); // 0, 1, 4, 9, 16, 25
}

// square the counter variable i each time the loop runs,
// but increment by 2 each time with i+=2
for(let i = 0; i < 10; i+=2) {
   console.log(i ** 2); // 0, 4, 16, 36, 64, 100
}


### index

The position number of an item in an array, with the first item at index 0:

```js
let fruits = ['kiwi', 'pear', 'plum'];
console.log(fruits[0]); // kiwi
```

---

key

The name of a property is called the *key*. This object has keys of **name**, **age** and **retired**.
```

```
let guy = {
   name: "Bob",
   age: 40,
   retired: false
}
```

length

Called on an array, the *length* property returns the number of items in the array:

```
let fruits = ['kiwi', 'pear', 'plum'];
console.log(fruits.length); // 3
```

---

let

The **let** keyword is for declaring block-scoped variables, meaning that the variables only exist inside the curly braces in which they are declared. In this example, **power** is declared in the **global scope** and so is available everywhere in the script. But **i** and **cube** are declared inside the for loop, which makes them **block scoped** variables, so NOT available in the global scope:

```
let power = 3;

for(let i = 0; i < 10; i++) {
   let cube = i ** power;
   console.log(cube);
}

console.log(power); // 3
console.log(i); // ERROR cube is not defined
console.log(cube); // ERROR cube is not defined
```

---

loop

A loop is a programming routine that executes a block of code again and again, as long as a condition is true. There are a few kinds of loops in JS, including "for loops" and "while loops":

```
   for(let i = 0; i < 10; i++) {
      console.log(i ** 4); // 1, 16. 81, 625 ... 10000
   }
```

---

method

Methods are the verbs of programming languages-—they do stuff; they perform actions. Just as verbs go together with nouns, methods are attached to objects.

- Methods perform a myriad of tasks: output content to html elements, load data from the server, play sounds, create new html elements, and so on.
- Dot-syntax is how we access the methods of a JS object. First comes the object, followed by a dot. Then comes the method. So it goes object-dot-method:

```
let dateTime = new Date();
// the Date object's getHours() method:
let hour = dateTime.getHours();
// the console's log method:
console.log(hour); // integer in 0-23 range
```

---

modulo

The **modulo operator**, symbol % returns the remainder when one number is divided by another:

```
console.log(15 % 4); // 3
```

---

nesting

Nesting refers to something inside of another similar thing, such as a loop inside of a loop, a block of conditional logic (if-statement inside) conditional logic, or an array inside of an array:

```
// nested for loop: a loop inside a loop
for(let i = 0; i < 5; i++) {
    for(let j = 0; j < 5; j++) {
        console.log(i**j); // i to the j power
    }
}

// nested if-else:
let weather = "cloudy";
let windy = true;

if(weather == "cloudy") {
    console.log("Go to the park");
    if(windy) {
        console.log("Fly a kite");
    } else {
        console.log("Have a picnic");
    }
}
```

---

objects

Objects are variables that can store more than one value at a time. They are divided broadly into Objects, per se, and a type of object called an Array. Objects proper store data inside curly braces as **properties**, consisting of name-value (key-value) pairs: name:

```
let cat = {
  name: "Fluffy",
  age: 4,
  cute: true
};
```

---

property

A property is a name-value (key-value pair) that belongs to—is scoped to—an object. As such, a property is essentially a local variable. The properties of an object can be nested. Properties are accessed by means od **dot syntax**:

```
let cat = {
  name: "Fluffy",
  age: 4,
  cute: true,
  owner: {
    name: "Bob",
    age: 40,
    job: "Programmer"
  }
};
console.log(cat.name); // Fluffy
console.log(cat.owner.name); // Bob
console.log(cat.age); // 4
console.log(cat.owner.age); // 40
console.log(cat.cute); // true
console.log(cat.owner.cute); // undefined
console.log(cat.job); // undefined
console.log(cat.owner.job); // Programmer
```

---

return value

Think of a function as an input-output machine, with the arguments being the input and the **return** value being the output. A **return** value “exports” the function result for use elsewhere. In order to "capture" the return value, set the function call equal to a variable. Many JS methods return a value.

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

let sum1 = addNums(5, 6);
console.log(sum1); // 11

let sum2 = addNums(8, 9);
console.log(sum2); // 17

// JS methods that return a value:
// instantiate the Date object
let dateTime = new Date();
// get the current hour as an integer from 0-23
let hour = dateTime.getHours(); // the hour from 0-23 is returned
console.log(hour); // 15 (if current time is 3:00-3:59pm)
// declare an array
let cars = ['Audi', 'BMW', 'Chevy', 'Dodge'];
// pop (remove) the last item; the pop() method returns the item
let lastCar = cars.pop();
console.log(lastCar); // Dodge
```

---

string

Pure text in quotes is a string. Setting a variable equal to a string gives the variable a data type of **string**:

```
let firstName = "Brian";
console.log(firstName, typeof(firstName)); // Brian string
```

---

ternary

A ternary expression is a concise, one-line alternative to an else-if statement. It uses **?** and **:** instead of **if()** and **else**, while also dispensing with the curly braces:

```
let x = 5;

// if x is less than 10, add 1; else set x equal to 0:
if(x < 10) {
   x++;
} else {
   x = 0;
}

// ternary version of the above:
x < 10 ? x++ : x = 0;
```

---

this

The this keyword refers to various objects, depending on where it is encountered:

- in the global scope, *this* is the Global Window Object
- inside a function, *this* is the object (button, menu, etc) whose event ('click', 'change', etc) called the function.
- in a function defined inside an object (a method), *this* refers to the object itself.

---

var

The **var** keyword for declaring variables has largely been supplanted by **let** due to the greater control over variable scope provided by the latter. A **var** declared inside the curly braces of a loop or if-statement are in the global scope, whereas **let** variables declared inside the curly braces of a loop or if-statement are scoped to that code block:

```
// j exists outside the loop, because it is declared with var:
for(var j = 0; j < 5; j++) {
   console.log(j**2); // j squared
}
console.log(j); // 5

// i exists ONLY inside the loop, because it is declared with let:
for(let i = 0; i < 5; i++) {
   console.log(i**2); // i squared
}
console.log(i); // ERROR: i is not undefined
```

---

while

A **while loop** is a kind of loop. Unlike a **for loop**, which has the counter, condition and incrementer inside parentheses, a while loop has these elements scattered about, with the counter declared above the loop, the condition evaluated inside

parentheses, and the incrementer inside the curly braces. In general, use a for loop if you know exactly how many times you want the loop to run, and use a while loop if the number of iterations is not pre-determined:

```
let frutes = ['apple', 'banana', 'grape', 'kiwi', 'orange', 'pear'];

// using a for loop, because the number of iterations is known:
for(let i = 0; i < frutes.length; i++) {
   console.log(frutes[i] + " has " + frutes[i].length + " letters!");
}

/* using a while loop because the number of iterations is not known; the loop ends with the first 4-letter item, but its index may not be known. */

let n = 0;
while (frutes[n].length != 4) {
   console.log(frutes[n] + " is " + frutes[n].length + " letters long!");
   n++;
}

/* the for loop will run 6 times as it fully traverses the array, but the while loop will only run 3 times, because it will quit when the condition is false, which occurs when the current item is 'kiwi'.
apple has 5 letters!
banana has 6 letters!
grape has 5 letters!
kiwi has 4 letters!
orange has 6 letters!
pear has 4 letters!
apple is 5 letters long!
banana is 6 letters long!
grape is 5 letters long!
*/
```

---

window

The window object, representing the browser, is at the top of the DOM hierarchy. It is the top-level object in JavaScript. It is above the document (web page) itself, which is a child of the browser. The console is another direct child of the window.