Modern Javascript

ITJ BOOTCAMP

Hello!

My name is
Isaac Cisneros

I am a Junior Web Developer at iTjuana.

Let's get started!

What is functional programming?

Functional programming is a style of programming that emphasizes the use of functions and immutable data.

In general, the following concepts are emphasized in functional programming:

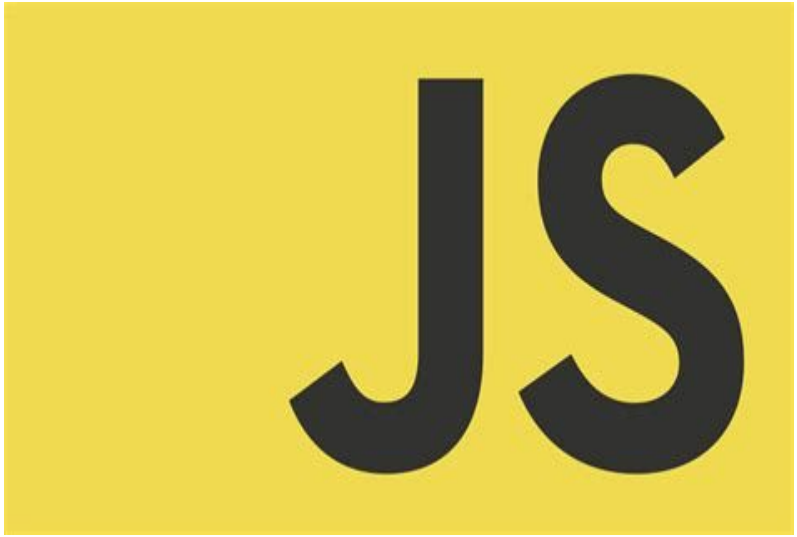
- Functions as the primary constructs you use
- Expressions instead of statements
- Immutable values over variables
- Declarative programming over imperative programming

What is Javascript?

JavaScript (often shortened to JS) is a lightweight, interpreted, object-oriented language with first-class functions, and is best known as the scripting language for Web pages, but it's used in many non-browser environments as well.

It is a prototype-based, multi-paradigm scripting language that is dynamic, and supports object-oriented, imperative, and functional programming styles.

JavaScript runs on the client side of the web, which can be used to design / program how the web pages behave on the occurrence of an event. JavaScript is an easy to learn and also powerful scripting language, widely used for controlling web page behavior.



Primitive Data Types

Data Type	Definition	Example
String	A JavaScript string is zero or more characters written inside quotes.	“Hello from the Fullstack bootcamp”
Number	JavaScript has only one type of number. Numbers can be written with or without decimals.	2022
Boolean	A JavaScript Boolean represents one of two values: true or false.	true
null	The value null represents the intentional absence of any object value.	null
undefined	In JavaScript, a variable without a value, has the value undefined. The type is also undefined.	undefined

Variable declaration statements

Statement	Description	Code example
let	Variables defined with let: ● Must be Declared before use. ● Cannot be Redeclared. ● Have Block Scope.	let greeting = “Hello there.”;
const	Variables defined with const: ● Cannot be Redeclared. ● Cannot be Reassigned. ● Have Block Scope.	const g = 9.81;

Logical Operators

Operator	Description
==	Equal to
===	Equal value and equal type
!=	Not equal
!==	Not equal value or not equal type
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

“Falsy” values

Value	Description
false	The keyword false
0	The number zero
-0	The number negative zero
“”, ‘’, ```	Empty string value
null	Null - the absence of a value
undefined	Undefined - the primitive value
NaN	NaN - not a number.

All values are considered “truthy” unless they are defined as “falsy”.

Type coercion

Type coercion is the automatic or implicit conversion of values from one data type to another (such as strings to numbers).

Here is a table with some examples:

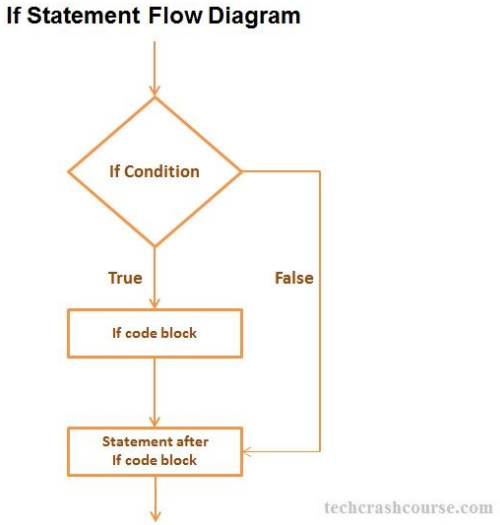
Original Value	Converted to Number	Converted to String	Converted to Boolean
“1”	1	“1”	true
“0”	0	“0”	true
0	0	“0”	false
“ ”	0	“ ”	false
[ ]	0	“ ”	true

Conditional statements

Conditional statements are used to perform different actions based on different conditions.

In JavaScript we have the following conditional statements:

- IF to specify a block of code to be executed, if a specified condition is true
- ELSE to specify a block of code to be executed, if the same condition is false
- ELSE IF to specify a new condition to test, if the first condition is false
- SWITCH to specify many alternative blocks of code to be executed



Conditional statements

Use the if statement to specify a block of JavaScript code to be executed if a condition is true.

Code Example

```
if (condition1) {
  //  block of code to be executed if condition1 is true
} else if (condition2) {
  //  block of code to be executed if the condition1 is false and condition2 is true
} else {
  //  block of code to be executed if the condition1 is false and condition2 is false
}
```

Conditional ternary operator

The conditional (ternary) operator is the only JavaScript operator that takes three operands: a condition, an expression to execute if truthy and an expression to execute if falsy.

The syntax is as following:

condition ? <expression if true> : <expression if false>

Code Example

```
authenticated ? renderApp() : renderLogin();
```

Functions

A JavaScript function is a block of code designed to perform a particular task and is executed when "something" invokes it (calls it).

Function parameters are listed inside the parentheses () in the function definition, arguments are the values received by the function when it is invoked. We can also declare default parameters.

Code Example

```
function functionExample (parameter1, parameter2) {
  // code to be executed
}

functionName(arg1, arg2, arg3); // function invocation
```

Arrow function expressions

An arrow function expression is a compact alternative to a traditional function expression, but is limited and can't be used in all situations.

For example, it does not have its own bindings to this or super, and should not be used as methods.

Code Example

```
const functionExample = (parameter1, parameter2) => {
    // Code to be executed
}

functionExample(arg1, arg2);
```

Objects

The Object class represents one of JavaScript's data types. It is used to store various keyed collections and more complex entities.

The following code example show how to create a new object:

Code Example

```
const person = { name: “Isaac”, age: 26, homeTown: “Tijuana” }
```

You can also declare function as property values, these functions are called methods.

Code Example

```
const person = {
   name: “Isaac”,
   sayHi: function () {
      console.log(“Hi”);
   }
}
```

Objects

You can access object properties in two ways:

Code Example
objectName.propertyName objectName["propertyName"]

You can assign and delete variables with the following syntax:

Code Example

person.nationality = "Mexican"; // assign or reassign a value
 	
delete person.nationality; // delete an existing value

Arrays

The Array object, as with arrays in other programming languages, enables storing a collection of multiple items under a single variable name, and has members for performing common array operations.

The following code example shows how to create an array using square bracket notation:

Code Example
const pets = ["Dog", "Cat", "Snake"];

You access an array element by referring to the index number:

Code Example
const myPet = pets[0];

Arrays

The easiest way to add a new element to an array is using the push() method:

Code Example

```
pets.push("Parrot");  // Adds a new element (Parrot) to pets
```

The splice() method coupled with indexOf() method removes a specific item or items from an array.

Code Example

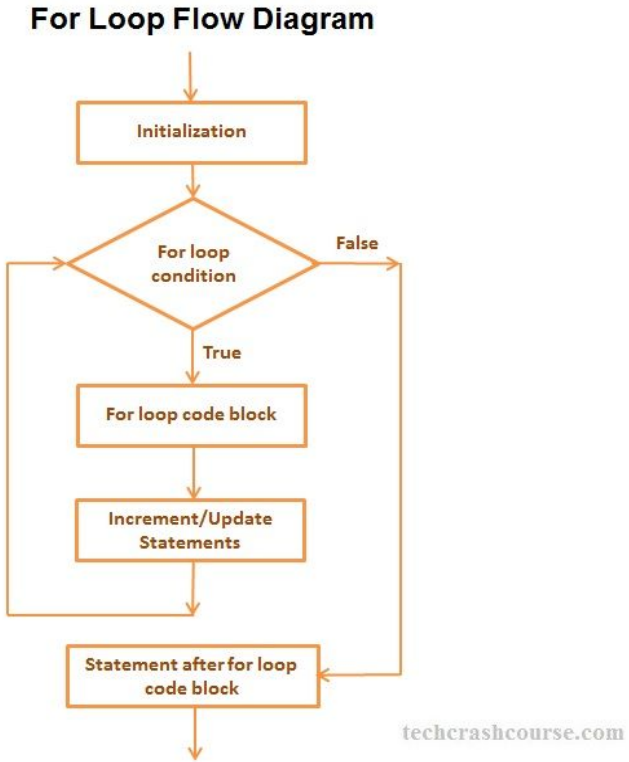
```
const start = pets.indexOf(“Snake”);
const deleteCount = 1;

fruits.splice(start, deleteCount);
```

Loops

Loops offer a quick and easy way to do something repeatedly. There are many different kinds of loops, but they all essentially do the same thing: they repeat an action some number of times.

The statements for loops provided in JavaScript are:

- for
- while
- do...while
- for..in
- for..of



Loops: for

A for loop repeats until a specified condition evaluates to false. The JavaScript for loop is similar to the Java and C for loop.

Code Example

```
for (let i = 0; i < 5; i++) {
  let text = "The number is " + i;
}
```

Spread syntax - Arrays

Spread syntax (...) allows an iterable such as an array expression or string to be expanded.

Code Example 1

```
const sum = (x, y, z) => {
  return x + y + z;
}
const numbers = [1, 2, 3];
console.log(sum(...numbers));
```

We can also merge two arrays using the spread syntax.

Code Example 1

```
const arr1 = [1,2,3];
const arr2 = [4,5,6];
const mergedArray = [...arr1, ...arr2];
```

Spread syntax - Object

With Rest/Spread syntax, you can spread properties of objects. It copies own enumerable properties from a provided object onto a new object.

Code Example 2

```
let obj1 = { product: “Chair”, cost: 299 };
let obj2 = { product: “Chair”, cost: 349, type: “Furniture” };

let clonedObj = { ...obj1 };

let mergedObj = { ...obj1, ...obj2 };
```

Rest syntax

The rest parameter syntax allows a function to accept an indefinite number of arguments as an array.

Code Example

```
const sum = (firstNumber, ...otherNumbers) => {
  console.log(firstNumber);
  console.log(otherNumbers);
};

console.log(sum(1, 2, 3));
console.log(sum(1, 2, 3, 4));
```

Object & Array manipulation

The forEach() method calls a function for each element in an array, and it is not executed for empty elements.

Code Example

```
const array = ["firstElement", "secondElement", "thirdElement"];

array.forEach((element, i) => {
  console.log(element);
  console.log(i);
});
```

Object & Array manipulation

The map() method creates a new array from calling a function for every array element, calls a function once for each element in an array, does not execute the function for empty elements, does not change the original array.

Code Example

```
// adds dollar sign to numbers

const numbers = [10, 3, 4, 6];
const dollars = numbers.map( number => '$' + number);
```

Object & Array manipulation

The filter() method creates a new array with all elements that pass the test implemented by the provided function.

Code Example

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

const result = words.filter(word => word.length > 6);
```

Object & Array manipulation

The reduce() method executes a reducer function for array element, returns a single value: the function's accumulated result, does not execute the function for empty, does not change the original array.

Code Example

```
const numbers = [100, 300, 500, 70];

const sum = numbers.reduce((accummulator, value) =>
accummulator + value
, 0);
```

Object & Array manipulation

The Javascript Object values() method retrieves an array of direct enumerable property values.

Code Example

```
const user = {
  age: 26,
  mobile: 8801967402131,
  name: "Jose"
}
const user = Object.values(user);
```

The entries() method returns a new Array Iterator object that contains the key/value pairs for each index in the array.

Code Example

```
const user = Object.entries(user);
```

Object & Array manipulation

The Object.keys() method returns an array of a given object's own enumerable property names, iterated in the same order that a normal loop would.

Code Example

```
const user = Object.keys(user1);
```

Nullish coalescing operator

The nullish coalescing operator (??) is a logical operator that returns its right-hand side operand when its left-hand side operand is null or undefined, and otherwise returns its left-hand side operand.

Code Example 1

```
const name = null ?? 'Isaac';
const lastName = undefined ?? 'Isaac';
```

Code Example 2

```
const number = 0 ?? 26;
const animal = “” ?? “”Crocodile;
```

Optional chaining operator

The optional chaining operator (?.) enables you to read the value of a property located deep within a chain of connected objects without having to check that each reference in the chain is valid.

Code Example

```
const person = {
  name: "Isaac Cisneros",
  socialMedia: {
    linkedIn: "@myLinkedIn",
    youtube: "@myYoutubeChannel",
  },
  createInstagram: function () {
    console.log("Instagram created.");
  },
};

// non existing method, wont give an error because of optional chaining.
person.createFacebook?.();

//non existing property, wont give an error because of optional chaining.
console.log(person.socialMedia?.TikTok);
```

Destructuring assignment

The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

The following example shows array and object destructuring:

Code Example 1

```
const user = {
    id: 42,
    isVerified: true
};

const {id, isVerified} = user;
```

Code Example 2

```
const numbers = ['one', 'two', 'three'];
// name does not matter when destructuring arrays
const [red, yellow, green] = numbers;
```

Async await

The async function declaration specifies an asynchronous function, which can return an AsyncFunction object. Async functions perform in a separate order than the rest of the code through the event loop and return a Promise as its result.

Code Example 1

```
const getData = async function () {
  const response = await fetch("https://jsonplaceholder.typicode.com/users");
  const data = await response.json();
  console.log(data);
};

getData();
```

Imports

The static import statement is used to import read only live bindings which are exported by another module.

Here are some import examples:

Code example 1 - Importing default exports from another module
import defaultExport from "./module1";
Code example 2 - Import named exports from another module:
import { export1 } from "./modules/module2";
Code example 3 - Import named exports from another module and give them an alias:
import { export1 as alias1 } from "../module/module3";

Exports

The export statement is used when creating JavaScript modules to export live bindings to functions, objects, or primitive values from the module so they can be used by other programs with the import statement.

The following are some examples of exports:

Code example - Named export
export { myFunction, country }; export let country = “Mexico”
Code example - Default export
export { myFunction as default }; export default country

Export and require - node

Node.js follows the CommonJS module system, and the builtin require function is the easiest way to include modules that exist in separate files. The basic functionality of require is that it reads a JavaScript file, executes the file, and then proceeds to return the exports object

Code example - File1 - Export

```
const printMessage = (message) => {
  console.log(message);
  return;
};

const message = “This is a message!”

module.exports = {
message,
printMessage,
}
```

Code Example - File2 - Require

```
const Examples2 = require("./examples2");
const message = Examples2.message;

Examples2.printMessage(message);
```