JavaScript

Table Of Content

Module 1 - JavaScript Basics

Your first "hello world" program

- Write the below HTML code in `index.html` file and open it in browser

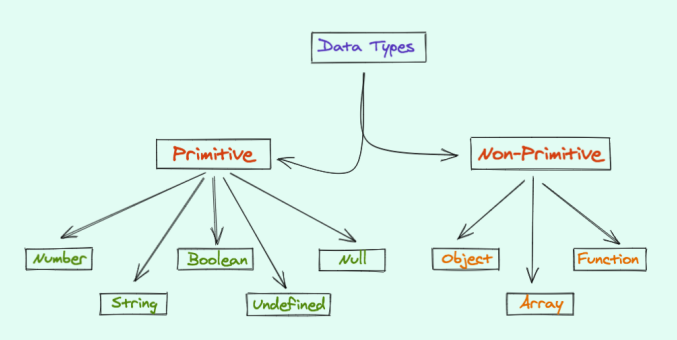
```
<!DOCTYPE html>
<html>
  <body>
    <h1>My First Web Page</h1>
    <script>
      console.log("Hello World");
    </script>
  </body>
</html>
```

- JavaScript code is written in between the `script` tag in the above code.
- When the page loads the browser will run the code between the `script` tag.
- `alert()` function will be called which will create a model with `hello world` text on it.

> Congratulation! You just wrote your first JavaScript program

Data Types in JavaScript

- Values used in your code can be of certain type - number or string for example
- This type is called data type of the language
- Data Types supported in JavaScript are: `Number, String, Boolean, Function, Object, Null, and Undefined`
- They are categorized as primitive or non-primitive data types
- Check the illustration below



- Unlike Java or C#, JavaScript is a loosely-typed language
- No type declarations are required when variables are created
- Data Types are important in a programming language to perform operations on the variables

```
// Data Types examples

var x = 10 // number variable
var x = "hi" // string variable
var x = true // boolean variable
function x { // your function code here }  // function variable
var x = { }  // object variable
var x = null  // null variable
var x  // undefined variable
```

Module 2 - Conditionals and Collections

If Else Condition

```
var x = 10;

if(x == 10) {
  console.log("x is 10")
}
else if(x < 10) {
  console.log("x is less than 10)
}
else {
  console.log("x is greater than 10)
}
```

- `if` block is executed if the condition is true
- `else if` block is used to specify additional conditions if the `if` condition is not satisfied
- `else` block is executed if neither of the prior conditions is satisfied

`truthy` and `falsy` values in JavaScript

- Boolean data types are either `true or false`
- But in JS in addition to this, everything else has inherent boolean values
    - They are `falsy` or `truthy`

- Following values are always `falsy`:

```
// falsy values

false
0 (zero)
"" (empty string)
null
undefined
NaN (a special Number value meaning Not-a-Number)
```

- All other values are `truthy`

```
// truthy values

"0" // zero in quotes
"false" // false in quotes
function () {} // empty functions
[] // empty arrays
{} //empty objects
```

- This concept is important because the inherent values can then be used in conditional logic
- You don't have to do `if(x == false)` - you can just do `if(!x)`

```
if (x) {
  // x is truthy
}
else {
  // x is falsy
  // it could be false, 0, "", null, undefined or NaN
}
```

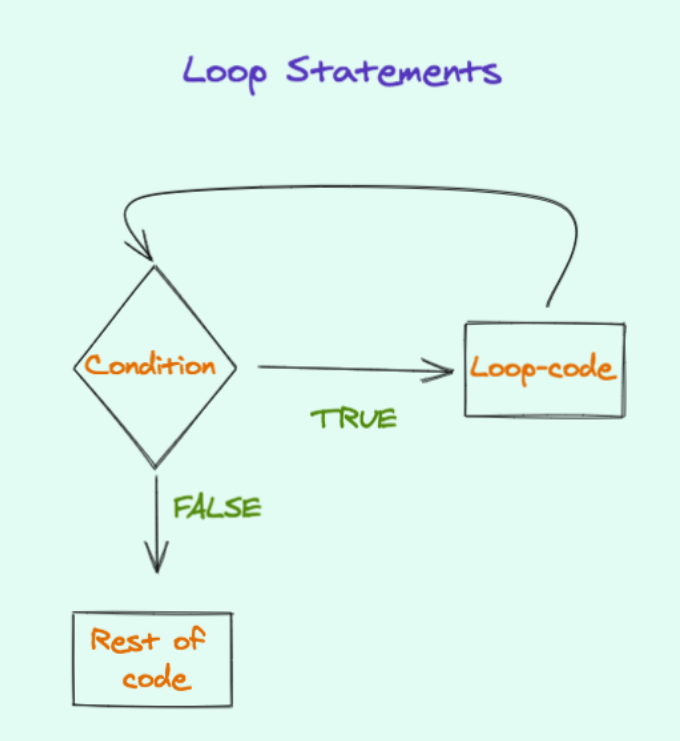
For Loop

- Loops are used to run the same code block again and again "for" given number of times

```
// ... your code

// This loop will be executed 10 times
for (i = 0; i < 10; i++) {
  console.log(i)
}

// ... your rest of the code
```



- Check out the illustration above
- It checks a condition first
- If the condition is true it will run the code inside the loop

- It will continue running the code inside the loop until the condition does not meet anymore
- After that the execution will come outside the loop and continue executing the rest of the code
- Loops come in handy when working with collections and arrays
- Below code will iterate over an array and log all its items

```
var items = [1,2,3,4]

for (i = 0; i < items.length; i++) {
  console.log(items[i]) // 1,2,3,4
}
```

Module 3 - JavaScript Objects and Functions

Function Expressions

- You can also create functions using another syntax
- You can assign an anonymous function to a variable, like below -

```
var addMe = function(a, b) {
  return a + b
}

var sum = addMe(1,2)
```

```
console.log(sum) // 3
```

- Please note that the name of the function is assigned to the variable instead of the function
- Result of the function remains the same

Scoping in JavaScript

- Every variable defined in JavaScript has a scope
- Scope determines whether the variable is accessible at a certain point or not

Two Types

- Local scope
  - Available locally to a "block" of code
- Global scope
  - Available globally everywhere

> JavaScript traditionally always had function scope. JavaScript recently added block scope as a part of the new standard. You will learn about this in the Advanced JavaScript module.

Examples

- Function parameters are locally scoped variables
- Variables declared inside the functions are local to those functions

```
// global scope
var a = 1;
```

```
function one() {
  console.log(a); // 1
}

// local scope - parameter
function two(a) {
  console.log(a); // parameter value
}

// local scope variable
function three() {
  var a = 3;
  console.log(a); // 3
}

one(); // 1
two(2); // 2
three(); // 3
```

Example: JavaScript does not have block scope

- In the below example value of `a` is logged as 4
- This is because JavaScript function variables are scoped to the entire function
- Even if that variable is declared in a block - in this case, the `if-block`
- This phenomenon is called as `Hoisting` in JavaScript

```
var a = 1

function four(){

  if(true){
    var a = 4
  }

  console.log(a) // logs '4', not the global value of '1'
```

```
  }
```

Module 4 - Prototypes and Prototypal Inheritance

JavaScript as Prototype-based language

- JavaScript does not contain "classes" that defines a blueprint for the object, such as is found in C++ or Java
- JavaScript uses functions as "classes"
- Everything is an object in JavaScript
- In JavaScript, objects define their own structure
- This structure can be inherited by other objects at runtime

What is a prototype?

- It is a link to another object
- In JavaScript, objects are chained together by prototype chain

```
Joe -> Person -> Object -> null
```

- JavaScript objects inherit properties and methods from a prototype

Example of Prototype

- Prototype property allows you to add properties and methods to any object dynamically

```
function Animal(name) {
  this.name = name
}

Animal.prototype.age = 10
```

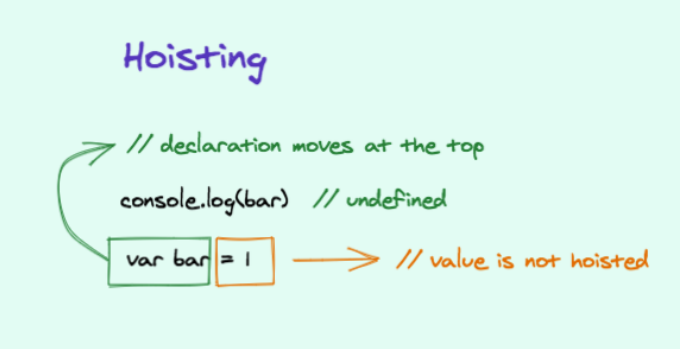
- When object `Cat` is inherited from object `Animal`
  - Then `Animal` is the prototype object or the constructor of the `Cat`

```
var Cat = new Animal('cat')
console.log(Cat) // constructor: "Animal"
console.log(Cat.name) // cat
console.log(Cat.age) // 10
```

Module 5 - Advanced JavaScript (Async JS, Closures, Method Chaining, etc.)

Hoisting in JavaScript

- In JavaScript function declarations and variable declarations are 'hoisted'
- Meaning variables can be used before they are declared



- From the illustration above - refer the code below
- We are logging `bar` variable to the console
- But, the variable `bar` is defined AFTER it is being used
- In other traditional languages - that would have been an error
- But, JavaScript does not throw any error here
- But, remember - the value of the variable is still `undefined` because the value is really assigned on AFTER it is being logged

```
console.log(bar) // undefined - but no error

var bar = 1
```

Another example

```
// Function declarations

foo() // 1

function foo() {
  console.log(1)
}
```

- The variable declarations are silently moved to the very top of the current scope
- Functions are hoisted first, and then variables
- But, this does not mean that assigned values (in the middle of function) will still be associated with the variable from the start of the function

- It only means that the variable name will be recognized starting from the very beginning of the function
- That is the reason, `bar` is `undefined` in this example

```
// Variable declarations

console.log(bar) // undefined

var bar = 1
```

> NOTE 1: Variables and constants declared with `let` or `const` are not hoisted!

> NOTE 2: Function declarations are hoisted - but function expressions are not!

```
// NO ERROR

foo();

function foo() {
  // your logic
}
```

We get an error with Function Expressions

- `var foo` is hoisted but it does not know the type `foo` yet

```
foo(); // not a ReferenceError, but gives a TypeError
```

```
var foo = function bar() {
  // your logic
}
```

Asynchronous JavaScript

Callback Function

- These are functions that are executed "later"
- Later can be any action that you'd want to be completed before calling the the callback function
- Callback functions are passed as arguments to the outer function

Simple example

- In this example `greet()` is the outer function
- And `getName()` is the callback function
- We pass `getName()` function to the outer `greet()` function as a function argument
- The value from `getName()` callback function is then used in the outer function `greet()`

```
function getName() {
  return "Sleepless Yogi";
}

function greet(callbackFn) {
  // call back function is executed here
  const name = callbackFn();

  return "Hello " + name;
}
```

- This was a very basic example
- Callback functions are more often used in asynchronous programming

```
Asynchronous programming

- This is the type of programming where actions does not take place in a
predictable order
- Example: network calls
- When you make an HTTP call you cannot predict when the call will return
- Therefore your program needs to consider this asynchronism to out the
correct results
```

Example callback in asynchronous programming

- In the below example we define a callback function `printUser`
- This function depends on the variable `name`
- So, basically until we have value for the `name` variable we cannot print the value
- We then define `fetchAndPrintUser` function to fetch the user and then print the user's name
- We are simulating network call using `setTimeout` method
- Basically it means after `500 ms` we will have the name available
    - In real world this will be a network call to some user API that queries the user database for this information
- After we get the user's name
- We call the callback function `printUser` with the name value

```
function printUser(name) {
  console.log(name)
}

function fetchAndPrintUser(printCallbackFunction) {

  // simulate fake network call
  setTimeout(() => {
    const fakeUserName = 'Sleepless Yogi'
```

```
    // We call the callback function here
    printCallbackFunction(fakeUserName)
  }, 500)
}

// Execute the function to fetch user and print the user's name
fetchAndPrintUser(printUser)
```

Module 6 - Next Generation JS - ES6 and Beyond

JavaScript Classes

- Classes were introduced in ES6 standard
- Simple `Person` class in JavaScript
- You can define `constructor` inside the class where you can instantiate the class members
- Constructor method is called each time the class object is initialized

```
class Person {
  constructor(name) {
    this.name = name
  }
}

var john = new Person("John")
```

Class methods

- You can add your functions inside classes
- These methods have to be invoked programmatically in your code

```
class Person {
  constructor(name) {
    this.name = name
  }

  getName() {
    return this.name
  }
}

john.getName() // John
```

- JavaScript class is just syntactic sugar for constructor functions and prototypes
- If you use `typeof` operator on a class it logs it as `"function"`
- This proves that in JavaScript a class is nothing but a constructor function

```
example:
class Foo {}
console.log(typeof Foo); // "function"
```

Class vs Constructor function

- Below example demonstrates how to achieve the same result using vanilla functions and using new classes
- You can notice how using `class` make your code cleaner and less verbose
- Using `class` also makes it more intuitive and easier to understand for Developer coming from class-based languages like Java and C++

Using Function - ES5 style

```
var Person = function(name){
    this.name = name
}

var Man = function(name) {
  Person.call(this, name)
  this.gender = "Male"
}


Man.prototype = Object.create(Person.prototype)
Man.prototype.constructor = Man

var John = new Man("John")

console.log(John.name) // John
console.log(John.gender) // Male
```

Using Classes - ES6+ Style

```
class Person {
    constructor(name){
        this.name = name
```

```
    }
}


class Man extends Person {
    constructor(name){
        super(name)
        this.gender = "Male"
    }
}


var John = new Man("John")

console.log(John.name) // John
console.log(John.gender) // Male
```

`let` and `const` and Block scope

- `let` and `const` keywords were introduced in ES6
- These two keywords are used to declare JavaScript variables

```
let myFirstName = "NgNinja"

const myLastName = "Academy"

console.log(myFirstName + myLastName) // "NgNinjaAcademy"
```

- These two keywords provide Block Scope variables in JavaScript
- These variables do not hoist like `var` variables

> Remember: using `var` to declare variables creates a function scope variables

- These two keywords lets you avoid `IIFE`
- `IIFE` is used for not polluting global scope
- But, now you can just use let or const inside a `block – {}` - which will have same effect

`let`

- `let` keyword works very much like `var` keyword except it creates block-scoped variables
- `let` keyword is an ideal candidate for loop variables, garbage collection variables

Example of `let`

- `var x` declares a function scope variable which is available throughout the function `checkLetKeyword()`
- `let x` declares a block scope variable which is accessible ONLY inside the if-block
- So, after the if-block the value of `x` is again `10`

```
function checkLetKeyword() {
  var x = 10
  console.log(x) // 10

  if(x === 10) {
    let x = 20

    console.log(x) // 20
  }

  console.log(x) // 10
}
```

`const`

- `const` keyword is used to declare a constant in JavaScript
- Value must be assigned to a constant when you declare it
- Once assigned - you cannot change its value

```
const MY_NAME = "NgNinja Academy"

console.log(MY_NAME) // NgNinja Academy

MY_NAME = "JavaScript" // Error: "MY_NAME" is read-only
```

Tricky `const`

- If you defined a constant array using `const` you can change the elements inside it
- You cannot assign a different array to it
- But, you can add or remove elements from it
- This is because `const` does NOT define a constant value. It defines a constant reference to a value.
- Example below:

```
const MY_GRADES = [1, 2, 3]

MY_GRADES = [4, 4, 4] // Error: "MY_GRADES" is read-only

MY_GRADES.push(4) // [1, 2, 3, 4]
```