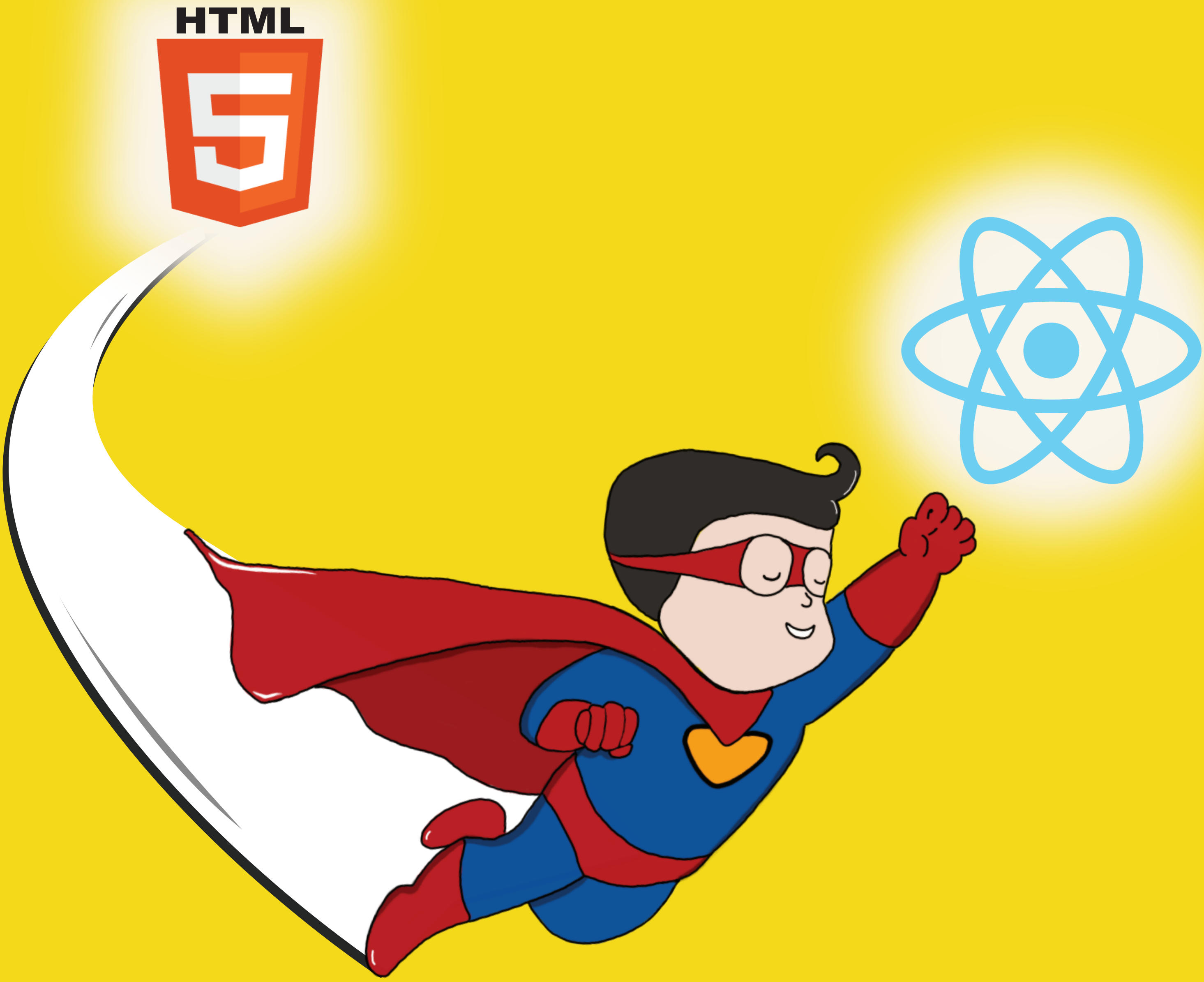


HTML TO REACT

The Ultimate Guide

NG NGNINJA ACADEMY

JavaScript

Table Of Content

Module 1 - JavaScript Basics

Who created JavaScript?

- Brendan Eich when working at NetScape
- It was created in 10 days

What is JavaScript

- JavaScript is an interpreted language
  - It means it doesn't need a compiler
  - It executes instructions directly without compiling
  - It is platform independence, dynamic typing
  - The source code is evaluated JUST before executing it
- It is open-source and cross-platform compatible
- It is created by NetScape
- It has object-oriented capabilities

Why do you love JavaScript?

- It is easy to start using
- JavaScript can be used on any platform
- It performs well on every platform
- You can build web, IOT, mobile apps using JavaScript

- It can be used on the Frontend, Backend, and also in the databases like MongoDB
- It is dynamic in nature ex: objects and arrays can be of mixed types

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

Run just JavaScript

- Instead of creating your own HTML file you can use online IDE as a JavaScript playground

- My favorite ones are:

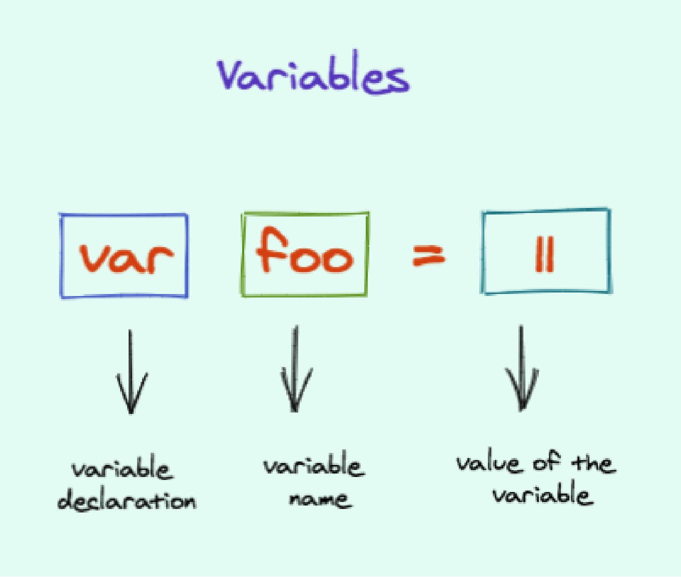
    - Code Sandbox
    - PlayCode

- Or you can also run JavaScript programs in VSCode
    - You need too install Node on your machine
    - Run `hit cmd + shift + p` on Mac, `ctrl + shift + p` on Windows / Linux
    - Type "Tasks: Configure Task"
    - Type "echo"
    - And replace your `task.json` file with below code
    - Then everytime you want to JavaScript program hit `hit cmd + shift + p` on Mac, `ctrl + shift + p` on Windows / Linux
    - Type "Tasks: Run Task"
    - Type "Show in console"

```
// task.json

{
  // See https://go.microsoft.com/fwlink/?LinkId=733558
  // for the documentation about the tasks.json format
  "version": "2.0.0",
  "tasks": [
    {
      "label": "echo",
      "type": "shell",
      "command": "echo Hello"
    },
    {
      "label": "Show in console",
      "type": "shell",
      "osx": {
        "command": "/usr/local/opt/node@10/bin/node ${file}"
      },
      "group": {
        "kind": "build",
        "isDefault": true
      }
    }
  ]
}
```

Variables

- Variables are containers
- They store data values
  - For ex: `var x = 5`
  - 5 is the value stored in variable `x`
- In programming, just like in mathematics, we use variables to hold values

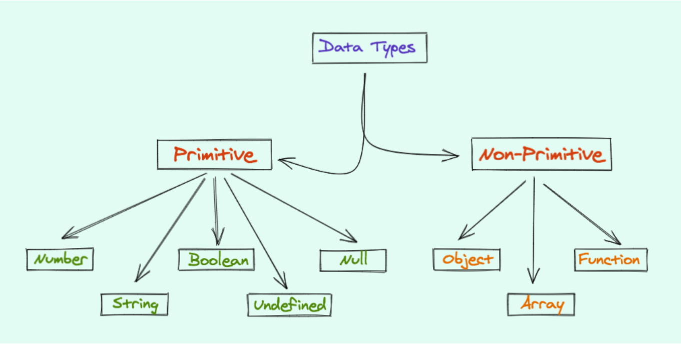


- Look at the illustration above
- `var` is the keyword used to declare a variable
  - In JavaScript you can also use `const` or `let`
  - If you don't use any keyword - the variable will be declared in a global scope
- `foo` is the name of the variable
  - Basically you give a name to some value
- `11` is the value you are storing in variable `foo`
  - This value can be anything you'd like
  - number, string, object, function - anything

```
// More examples

var x = 10 // number variable
var x = "hi" // string variable
```

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
```

Basic Operators

- `=` operator is used to assign value to a variable
  - ex: `var x = 10` - variable `x` is assigned value `10`
- `+` operator is used to add numbers
  - ex: `var x = 10 + 5` - variable x is now 15
- `+` operator is also used to concatenate two strings
  - ex: var `x = "hi" + "there"` - variable `x` is now `hithere`
- `-` operator is used to subtract numbers
  - ex: var `x = 10 - 5` - variable `x` value is now `5`
- `*` operator is used to multiple numbers
  - ex: var `x = 10 * 5` - variable `x` value is now `50`
- `/` operator is used to divide numbers
  - ex: var `x = 10 / 5` - variable `x` value is now `2`
- `++` operator is used to increment value of the variable
  - ex: `var x = 10; x++;`- variable `x` value is now `11`
- `--` operator is used to decrement value of the variable
  - ex: `var x = 10; x--;` - variable `x` value is now `9`

Special Operators

- `typeof` operator can be used to return the type of a variable
  - Use `typeof` for simple built in types
- `instanceof` operator can be used to check if the object is an instance of a certain object type
  - Use `instanceof` for custom types

```
'my string' instanceof String; // false
typeof 'my string' == 'string'; // true

function() {} instanceof Function; // true
typeof function() {} == 'function'; // true
```

Fun with Operators

```
1.

var x = 15 + 5 // 20
var y = "hi"

var z = x + y // 20hi
```

```
2.

var y = "hi" + 15 + 5 // hi155
```

- In the first example
  - `15 + 5` is treated as number operation
  - When the compiler sees `hi` it performs string concatenation
  - So the answer is `20hi`
- In the second example
  - JavaScript compiler sees `hi` string first so it considers the operands as strings
  - So the answer is string concatenation `hi155`

JavaScript as Object-Oriented Programming language

- JavaScript has OOP capabilities like `Encapsulation, Aggregation, Composition, Inheritance, and Polymorphism`
- Aggregation
    - `A "uses" B` = Aggregation : B exists independently (conceptually) from A
    - example:
        - Let say we have objects: `address, student, teacher`
        - We want to specify `student address` and `teacher address`
        - Then we can reuse `address` between `student` and `teacher`
- Composition
    - `A "owns" B` = Composition : B has no meaning or purpose in the system without A
- Inheritance
    - Inheritance can be implemented in JavaScript like below
    - `class Car { }`
    - `class Honda extends Car { }`
- Douglas Crockford says - "In its present form, it is now a complete object-oriented programming language."
    - `http://JavaScript.crockford.com/JavaScript.html`

Polymorphism Example in JavaScript

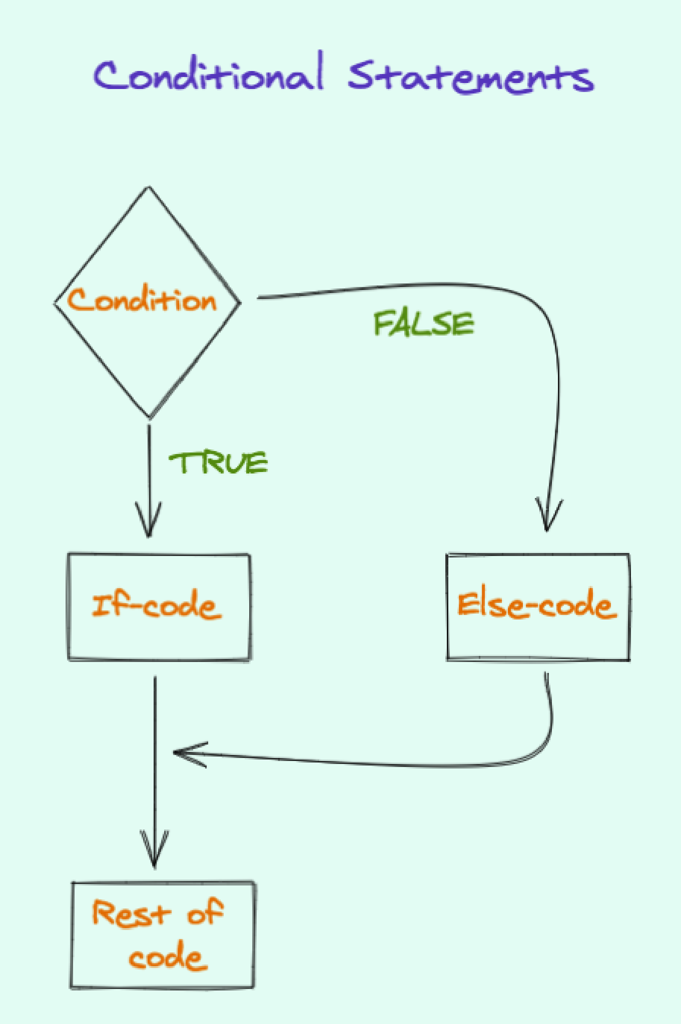
- We have two classes `Car` and `Bike`
- Both are Vehicles. Both vehicles `move`.
- But depending on the type of the vehicles they move differently
    - ex: `Car` drives
    - ex: `Bike` rides
- But from the user's point of view they just have to call `move()` method
- And depending on the type the respective objects will take care of calling the appropriate methods underneath.

```
class Car {
  constructor(vehicle) {
    this._vehicle = vehicle;
  }

  move() {
    console.log("drive", this._vehicle);
  }
}

class Bike {
  constructor(vehicle) {
    this._vehicle = vehicle;
  }

  move() {
    console.log("ride", this._vehicle);
  }
}

function getVehicle(vehicle) {
  switch (vehicle.type) {
    case "bike":
      return new Bike(vehicle);
    case "car":
      return new Car(vehicle);
    default:
      break;
  }
}

// this would create the appropriate vehicle using the above classes
let vehicle = getVehicle({
  type: "bike",
});

vehicle.move(); // ride { type: 'bike' }

vehicle = getVehicle({
  type: "car",
});

vehicle.move(); // drive { type: 'car' }
```

Module 2 - Conditionals and Collections

Conditionals

- You can make decisions in your code using conditional statements
- Essentially, they let you write -> `if this is true, do this - else do that`



- The above flow chart can be represented as below in the JavaScript code

```
// ...your code

if(some-condition == true) {
  // execute some code
}
else {
  // execute some other code
}

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

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

Ternary Operator

- `if-else` block can be simplified and written in lesser verbose code

```
// using if else

if(x == 10) {
  console.log("x is 10")
}
else {
  console.log("x is NOT 10")
}


// using ternary

x == 10 ? console.log("x is 10") : console.log("x is NOT 10")
```

- `condition ? if-code : else-code` is the syntax used for the ternary operator

Advanced Ternary

- You can also nest the ternary operators if there are complex conditions

```
// using if else

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

```
// using nested ternary

x == 10 ? (x == 10 ? console.log("x is 10") : console.log("x is less than
10") )  : console.log("x is greater than 10")
```

- `condition ? nested-ternary : else-code` - this is the syntax we used for the above-nested ternary operation
- You can go multiple levels deep into writing nested ternary operator
- But it is recommended to keep the ternary operators as simple as possible to keep the code more readable

Switch Statements

- It is another way to write conditional statements
- Based on conditions it can perform different actions

```
switch(x) {
  case 10:
    console.log("x is 10")
    break
  case 20:
    console.log("x is 20")
    break
  default
    console.log("x is NOT 10 nor 20")
}
```

- `switch(x)` this is where you specify the condition to be evaluated
- `case 10:` this is where you specify if the result of the condition equals this value the block of code will be executed
- `break` statement is required to break out of `switch` block.
  - If not provided it will execute all the following cases until it hits a `break` keyword or until the `switch` block is executed completely.
- `default` case is executed if none of the prior case conditions are met
  - `default` case does not have to be the last case in a switch block
  - `default` case is not required

`truthy` and `falsy` values in JavaScript

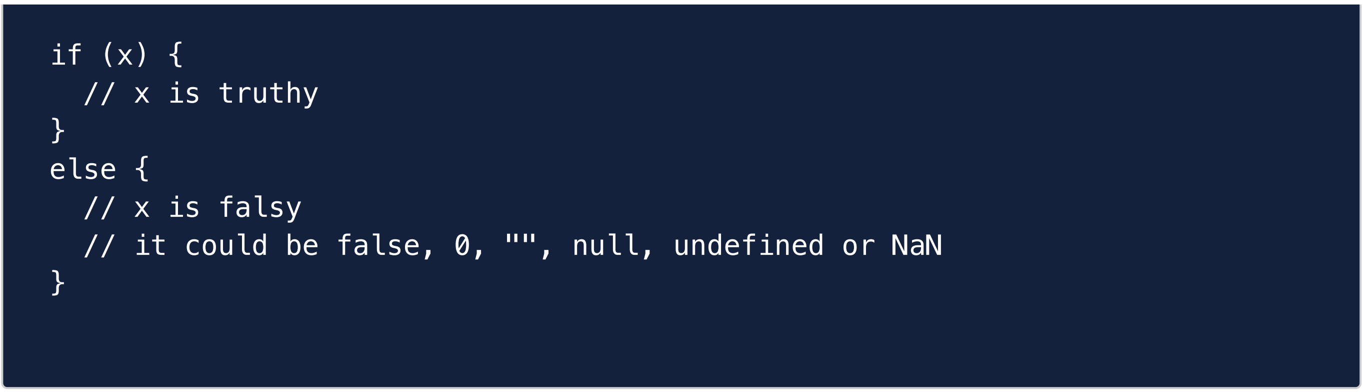
- Boolean data types are either `true or false`
- But in JS in addition to this, everything else has inherent boolean values
    - They are `falsy` or `truthy`
- Following values are always `falsy`:

```
// 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)`

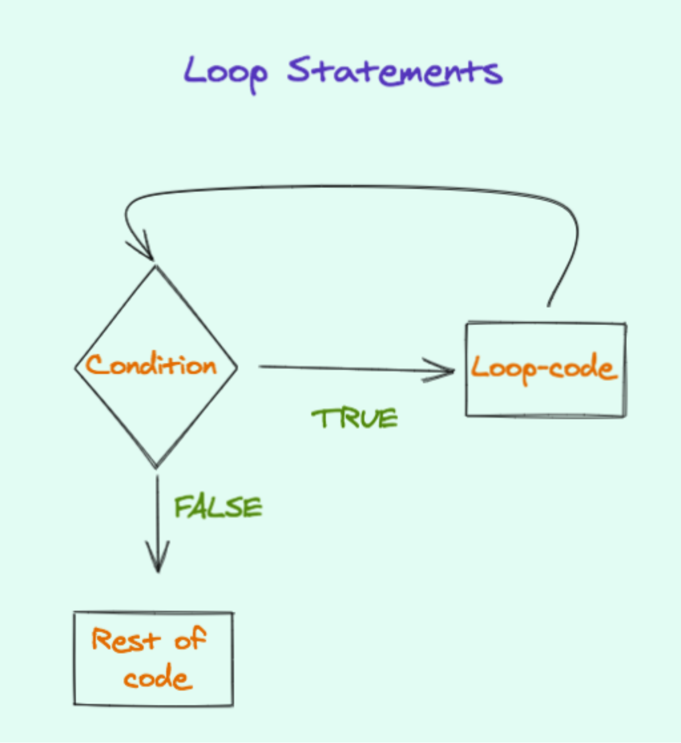


```
if (x) {
  // x is truthy
}
else {
  // x is falsy
  // it could be false, 0, "", null, undefined or NaN
}
```

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
}
```

For-In loop

- It is similar to `for` loop but is used to iterate over an object instead of an array

```
var myObject = {foo: "Dan", bar: 2};

for (var x in myObject) {

  // displays the object keys
  console.log(x) // foo, bar

  // displays the values of the keys
  console.log(myObject[x]) // Dan, 2

}
```

For-Of loop

- This kind of looping loops through the values of an iterable objects
- For ex: array or string
- You can directly use the values instead of using index on that array or the string

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


// using simple for loop

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

// using for-of loop

for(var x of items) {
  console.log(x) // 1, 2, 3
}
```

While loop

- This loop executed a block of code "while" the given condition is true

```
// This loop will be executed 10 times

var i = 0
while (i < 10) {
  console.log(i)
  i++
}
```

> NOTE: Remember to terminate the while condition properly. Or else the loop will go into infinity and it might crash your browser.

Do-While loop

- It is similar to the `while` loop except it executes the block of code first and then checks for the condition
- This process will repeat until the condition is true

```
// This loop will be executed 10 times

var i = 0
do {
  console.log(i)
  i++
} while (i < 10)
```

> Tip: In my experience, I have rarely used this `do-while`. Most of the time you can get away with using the `for` or the `while` loop.

Map Reduce Filter

Map

- It is used for creating a new array from an existing one
- It applies the given function to each item in that array

```
function getSquare(item) {
  return item * item
}

const numbers = [1, 2, 3, 4];
const squareOfNumbers = numbers.map(getSquare);
console.log(squareOfNumbers); // [1, 4, 9, 16]
```

- In the above example `getSquare` method is called for each item in the `numbers` array
- The method returns the square of each number
- The result of the `.map` is a new array with square of each number

Reduce

- Similarly to `.map` - `.reduce` calls the given method for each element in the array
- The result of each method call is passed over to the next method call in the array
- This result is called as `accumulator`
    - It can anything like a string, number or any object
- You can also pass in an `initial value` of the accumulator as an optional argument

```
function getSum(result, item) {
  return result + item
}

const numbers = [1, 2, 3, 4];
const sumOfNumbers = numbers.reduce(getSum, 0);
console.log(sumOfNumbers); // 10
```

- In the above example `getSum` method is called for each item in the `numbers` array
- `0` is passed as the initial value of the accumulator
- `result` is the variable name of the accumulator
- The above `.reduce` method adds each item in the array and stores that sum in the `result` variable
- Finally the `result` is returned to `sumOfNumbers`

Filter

- This method returns a subset of the given array
- It executes the given function for each item in the array and depending on whether the function returns `true` or `false` it keeps that element in or filters it out
- If `true` the element is kept in the result array
- If `false` the element is excluded from the result array

```
function isGreaterThanTwo(item) {
  return item > 2
}

const numbers = [1, 2, 3, 4];
var greaterThanTwoArray = numbers.filter(isGreaterThanTwo);
console.log(greaterThanTwoArray); // [3,4]
```

- In the above example `isGreaterThanTwo` method checks if the value of the given item is greater than two
- The result is a new array with only `[3,4]` items in it

Module 3 - JavaScript Objects and Functions

JavaScript Object Basics

- JS objects are used to represents real-life objects in most cases
    - Ex: Person, Vehicle, Monitor
- But, you can make an object for practically anything

```
const foo = {} // foo is an object
```

- Objects are variables
- They represent various attributes of a certain entity
- `person` object below represents a Person whose name is "foo" and age is 21
    - `name` is the property key
    - `foo` is the property value

```
const person = {
  name: "foo",
  age: 21
}
```

Access Object Value

- You can access object property value in two ways

```
1.
console.log(person.name) // foo


2.
console.log(person['age']) // 21
```

JavaScript Functions

- It is a piece of code ideally with a single purpose
- It is a wrapper around a piece of code
- It provides an abstraction to a block of code
- It provides a way to reuse functionality

Example Function

- Below is an example of JavaScript function
- `addMe` is the name of the function
- `a and b` are two arguments
    - JavaScript arguments are dynamic so you can pass it any value
- The function `addMe` returns the sum of two arguments `a and b`

```
function addMe(a, b) {
  return a + b   // The function returns the sum of a and b
}
```

Invoke Function

- Below is how you can invoke the `addMe` function
- `1 and 2` are arguments passed to the function which corresponds to `a and b` respectively
- The `return` value of the function is then stored in the variable `sum`
    - `return` statement is optional

```
let sum = addMe(1,2)
console.log(sum) // 3
```

Local variables

- You can define variables inside the function
- In the below example we have just passed in `a` variable
- The function `addMe` defines variable `b` inside the function
- Such variables like variable `b` are called local variables

```
function addMe(a) {
  let b = 2
  return a + b
}


let sum = addMe(4)
console.log(sum) // 6
```

- Local variables are not accessible outside the function

```
function addMe(a) {
  let b = 2
  return a + b
}
```

```
console.log(b) // ERROR - b is not defined
```

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'
  }
```

Constructor Functions

- Functions used to create new objects are known as constructor functions
- Below function `Person` is a standard function
- But the function is used to create a new object called `john`
- Therefore, the `Person` function by convention is called a constructor function

> It is considered good practice to name constructor functions with an upper-case first letter. It is not required though.

```
function Person() {
  this.name = "John"
  this.age = 21
}

var john = new Person()
```

The `this` keyword

- The `this` represents the object (or function) that "owns" the currently executing code.
- `this` keyword references current execution context.
- When a JavaScript function is invoked, a new execution context is created.
- `this` in js is different than other languages because of how functions are handled
  - Functions are objects in JavaScript
  - So we can change the value of `this` keyword for every function call

`this` with example

- The value of `this` depends on the object that the function is attached to
- In the below example;
    - `getMyAge` function belongs to `person` object
    - So, `this.age` represents the `person` object's `age` property

```
const person = {
  name: "foo",
  age: 21,
  getMyAge: function() {
    return this.age // 21
  }
}
```

More `this` examples

- Reference to the top-level execution context
- In the browser below `this` represents the `window` object

```
function go() { console.debug(this); }
go();
```

- In below example -

- `var foo = 10;` statement declares `foo` variable on the `window` object

- `print();` belongs to `window` object of browser

- So, `this.foo` returns the value of `foo` variable on the `window` object - which is `10`
- `var myObject = { foo : 20};` declares `foo` property which belongs to `myObject` object
- `print.apply(myObject);` statement simply makes `myObject` the owner of the `print` method
- So, `this.foo` now returns the value of `foo` variable on the `window` object - which is `20`

> NOTE: We will learn more about `apply` method in Module 5

```
var myObject = { foo : 20 };
var foo = 10;

function print(){
  console.log(this.foo);
}

// This will log window.foo - 10
print(); //

// This will alert myObject.foo which is 20
print.apply(myObject);
```

The new Operator

- It will create a new instance of an object
- It can be user-defined or a builtin type

```
// built-in type object

var cars = new Array('Honda', 'Audi', 'BMW');


// user-defined object

class Car {
  constructor(name) {
    this.name = name;
  }
}

var car = new Car('Honda')
```

> NOTE: You will learn about JavaScript Classes in Module 6

- It links the newly created object to another object
    - It does it by setting its constructor to another object
    - The object type is set to its constructor function
- It makes the `this` variable point to the newly created object.
- It invokes the constructor function

- `object.prototype` property is set to the object's prototype

Understand with example

- `Car` is a constructor function because it is invoked using `new` keyword
- `Car` function has a field called `name`
- `myCar` object is created from the `Car` function using `new` keyword
- When that is done:
    - It makes `Car` the prototype/constructor of `myCar`
    - It sets the `name` field to `Honda`
    - The value of `myCar` becomes `{name:'Honda'}`

```
function Car(name) {
  console.log(this) // this points to myCar
  this.name = name;
}

var myCar = new Car('Honda')
console.log(myCar) // {name: "Honda", constructor: "Car"}
```

Example of creating an object with and without `new` operator

WITHOUT new operator

- `this.A = 1;` - value of `this` is undefined so this statement will throw error
- `var t = Foo();` - value of `t` will be undefined because `Foo()` function is not returning anything

```
var Foo = function(){
  this.A = 1;
};

var t = Foo();
console.log(t); // undefined
```

WITH new operator

- `var m = Foo();` - value of `m` is `{ A: 1 }` with constructor set to `Foo`
- `this.A = 1;` - value of `this` is `m` object

```
var Foo = function(){
  this.A = 1;
};

var m = new Foo();
console.log(m); // m is { A: 1 }, type of m is Foo
```

Interview Question: What is the difference between the `new` operator and `Object.create` Operator

`new` Operator in JavaScript

- This is used to create an object from a constructor function
- The `new` keywords also execute the constructor function

```
function Car() {
  console.log(this) // this points to myCar
  this.name = "Honda";
}

var myCar = new Car()
console.log(myCar) // Car {name: "Honda", constructor: Object}
console.log(myCar.name) // Honda
console.log(myCar instanceof Car) // true
console.log(myCar.constructor) // function Car() {}
console.log(myCar.constructor === Car) // true
console.log(typeof myCar) // object
```

`Object.create` in JavaScript

- You can also use `Object.create` to create a new object
- But, it does not execute the constructor function
- `Object.create` is used to create an object from another object

```
const Car = {
  name: "Honda"
}

var myCar = Object.create(Car)
console.log(myCar) // Object {}
console.log(myCar.name) // Honda
console.log(myCar instanceof Car) // ERROR
console.log(myCar.constructor) // Anonymous function object
console.log(myCar.constructor === Car) // false
console.log(typeof myCar) // object
```

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

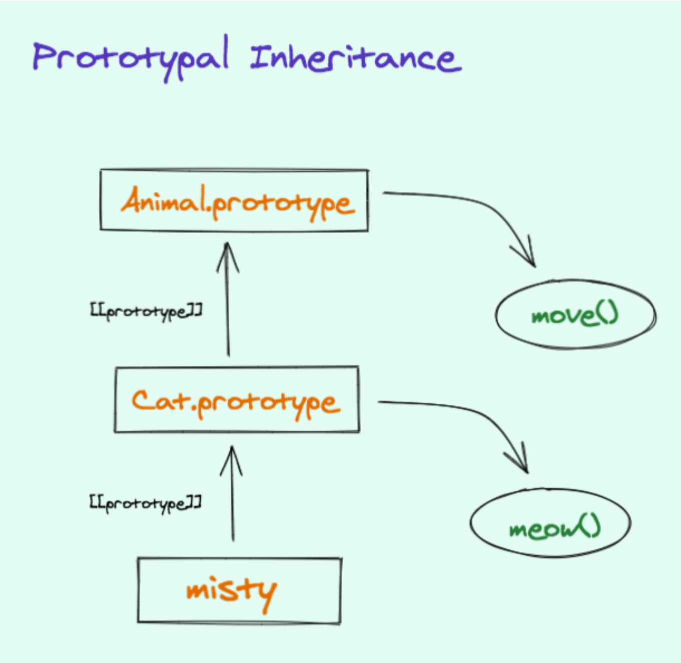
```
function Animal(name) {
  this.name = name
}

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
```

What is Prototypal Inheritance?

- In JavaScript object inherits from object - unlike class inheritance in C++ or Java
- Prototypal inheritance means that if the property is not found in the original object itself
    - Then the property will be searched for in the object's parent `prototype` object.
- Object literally links to other objects



- Check out the illustration above and refer the code below

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

Animal.prototype.move = function () {
  console.log("move");
};

function Cat(name) {
  Animal.call(this, name);
}

Cat.prototype.meow = function () {
  console.log("meow");
};
```

- `Animal` object is at the top of the inheritance (for this example)
- It has a `Animal.prototype` property on it
- We then have `Cat` object
- To execute a prototypal inheritance we have to link their prototypes
- Below is how you do it

```
Cat.prototype = Object.create(Animal.prototype)
```

- Now `Cat.prototype` is linked with `Animal.prototype`
- Then we create `misty` object from `Cat`

```
var misty = new Cat('misty')
```

- Now our new `misty` cat object will inherit all the properties on `Animal` and `Cat` object and also the properties on `Animal.prototype` and `Cat.prototype`

```
console.log(misty); // constructor: "Animal"
console.log(misty.name); // cat
console.log(misty.meow()); // meow
console.log(misty.move()); // move
```

Understand Prototypal Inheritance by an analogy

- You have exam, you need a pen, but you don't have a pen
- You ask your friend if they have a pen, but the don't - but they are a good friend
- So they ask their friend if they have a pen, they do!
- That pen gets passed to you and you can now use it
- The `friendship` is the `prototype` link between them!

Why is Prototypal Inheritance better?

- It is simpler
  - Just create and extend objects
  - You don't worry about classes, interfaces, abstract classes, virtual base classes, constructor, etc...
- It is more powerful
  - You can "mimic" multiple inheritance by extending object from multiple objects
  - Just handpick properties and methods from the prototypes you want
- It is dynamic
  - You can add new properties to prototypes after they are created
  - This also auto-adds those properties and methods to those object which are inherited from this prototype
- It is less verbose than class-based inheritance

Example of Prototypal Inheritance

```
function Building(address) {
  this.address = address
}
```

```
Building.prototype.getAddress = function() {
  return this.address
}

function Home(owner, address){
  Building.call(this, address)
  this.owner = owner
}

Home.prototype.getOwner = function() {
  return this.owner
}

var myHome = new Home("Joe", "1 Baker Street")

console.log(myHome)
// Home {address: "1 Baker Street", owner: "Joe", constructor: Object}

console.log(myHome.owner) // Joe
console.log(myHome.address) // 1 Baker Street
```

- Let's define accessor methods on the above constructor function
- `getAddress` method is defined on `Building`
- `getOwner` method is defined on `Home`

```
// On Building constructor
Building.prototype.getAddress = function() {
  return this.address
}

// On Home constructor
Home.prototype.getOwner = function() {
  return this.owner
}


var myHome = new Home("Joe", "1 Baker Street")

console.log(myHome.getOwner()) // Joe
console.log(myHome.getAddress()) // ERROR: myHome.getAddress is not a
function
```

- `getOwner` works correctly
- But - `getAddress` method gives `error`
- That is because we have not linked the `prototype` of `Home` to the `prototype` of `Building`

Linking the prototypes

- We can link the prototype by using `Object.create`
- Now when we call `getAddress` we get the value correctly as expected

```
Home.prototype = Object.create(Building.prototype)

console.log(myHome.getOwner()) // Joe
console.log(myHome.getAddress()) // 1 Baker Street
```

Prototype Chain

- In JavaScript, objects are chained together by a prototype chain
- If I the object don't have a property or method that is requested -
  - Then go to the object's prototype and look for it
- This process is repeated until JavaScript hits the top-level builtin object - `Object`

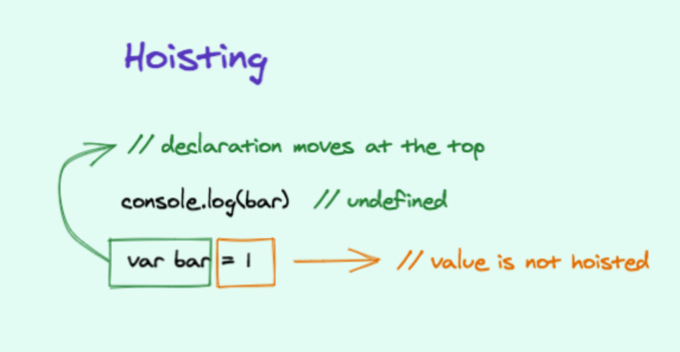
How does prototypal inheritance/prototype chain work in above example?

- JavaScript checks if `myHome` has an `getAddress` method - it doesn't
- JavaScript then checks if `Home.prototype` has an `getAddress` method - it doesn't
- JavaScript then checks if `Building.prototype` has an `getAddress` method - it does
- So, JavaScript then calls the `getAddress` on the `Building` function

Module 5 - Advanced JavaScript (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



```
// declaration moves at the top
console.log(bar)  // undefined
var bar = 1  // value is not hoisted
```

- 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
}
```

JavaScript Closures

> Technical Definition: Closure is when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope.

- Whenever you see a function keyword within another function, the inner function has access to variables in the outer function.
- That is a closure.

- Simply accessing variables outside of your immediate lexical scope creates a closure.
- Below example is a closure
- Because `a` is outside the scope of `function foo`

```
var a = 42;

function foo() { return a; }
```

- Closures are just using variables that come from a higher scope

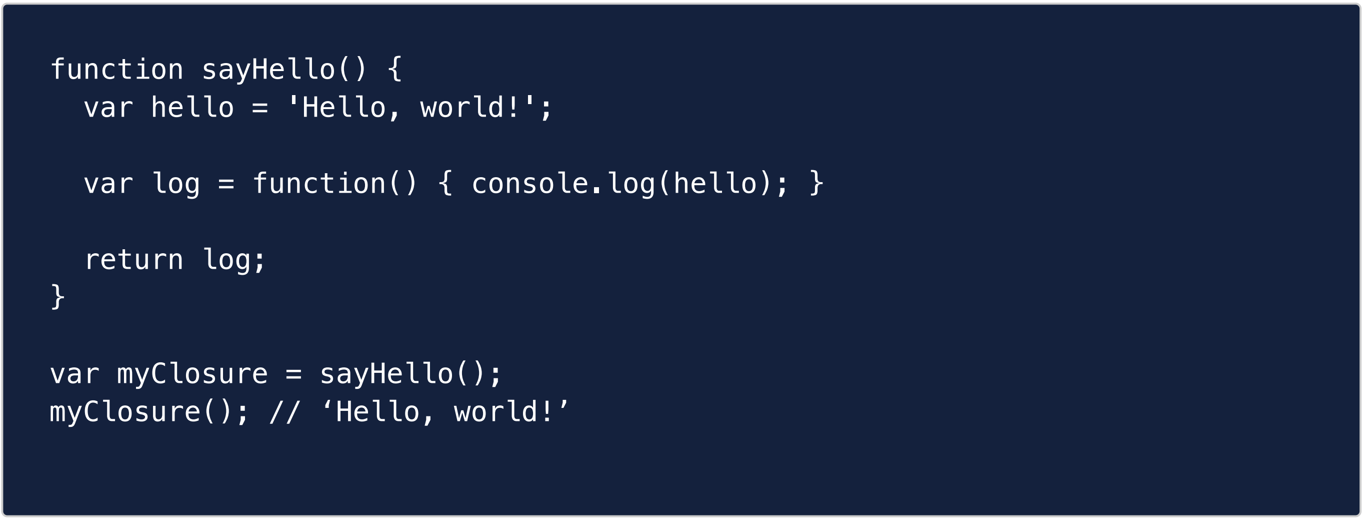
Closure remembers the environment

- The function defined in the closure 'remembers' the environment in which it was created
- Closure happens when an inner function is defined in outer function and is made accessible to be called later.

- In the below example we have a function `sayHello`
- It declares a local variable called `hello`
- It also declares a function variable called `log()`
- And finally, it returns the `log()` function

- So, `myClosure` variable is now pointing to the `log()` function
- Meaning calling `myClosure()` function is actually invoking the `log()` function from the `sayHello()` function

- And if you see the result - `log()` functions accurately logs the value of `hello` variable which was originally declared in the parent function `sayHello()`
- It means, the `log()` function has accurately "remembered" the value of the `hello` variable
- This phenomenon is called `closure`
- The value of `hello` variable is successfully locked into the closure of the `log()` function



```
function sayHello() {
  var hello = 'Hello, world!';

  var log = function() { console.log(hello); }

  return log;
}

var myClosure = sayHello();
myClosure(); // ‘Hello, world!’
```

IIFE

- It is called as Immediately Invoked Function Expressions

What is happening here?

```
(function foo(){

    // your code

})()
```

- It is `function expression`
- It is moreover a self-executing function - an `IIFE`
- It wraps the inside members to the scope
- It prevents from polluting the global scope
- It is useful in closures

Closure And IIFE

- `sum` is a Function expression whose value is an IIFE
- So, consequently, the `sum` is assigned the return value of a self-invoking function

```
var sum = (function() {
  var foo = 20

  function bar() {
    foo = foo + 10

    console.log(foo)
  }

  return bar

})()


sum() // 30
sum() // 40
sum() // 50
```

- What is happening inside IIFE?
- We have defined `foo` variable as the local variable inside the function
- We also have a function called `bar()`
- And, finally, we return the `bar` function
- So, the function `bar` is getting assigned to the variable `sum`

- What is happening inside the `bar()` function?
- We are accessing variable `foo` from the parent scope
- And we are incrementing its value by `10` and reassigning the new value back to the variable `foo` from the parent scope
- And finally, we are logging the new value of the variable `foo`

- The interesting part is, the value of `foo` is enclosed inside the IIFE which is assigned to `sum`
- And, `sum` is actually the function `bar` as you can see below
- Every time you call function `sum()` it updates and remembers the new value of variable `foo`
- Therefore, every call to the function displays the updated value of the `foo`

```
console.log(sum) // function bar() {}
```

JavaScript `call() & apply() vs bind()`?

- They all are used to attach a correct `this` to the function and invoke it
- The difference is the way of function invocation

`bind`

- It returns a function
- This returned function can later be called with a certain context set for calling the original function
- The returned function needs to be invoked separately

Example using `bind()`

- `person` object has a method called `hello()`
- `ngNinja` object does not have it
- You can bind `hello()` method to `ngNinja` object and call it later in the code

```
var person = {
  hello: function(message) {
    console.log(this.name + " says hello " + message)
  }
}

var ngNinja = {
  name: "NgNinja Academy"
}


var sayHello = person.hello.bind(ngNinja)
```

```
sayHello("world");  // output: "NgNinja Academy says hello world"
```

`call()`

- `call()` attaches `this` to function and invokes the function immediately
- The owner object is sent as an argument
- With `call()`, an object can use a method belonging to another object

- In the below example `this` is set to the `ngNinja` object
- You can send arguments to the function as a comma-separated list following the owner object

```
var person = {
  hello: function(message) {
    console.log(this.name + " says hello " + message);
  }
}

var ngNinja = {
  name: "NgNinja Academy"
}

person.hello.call(ngNinja, "world"); // output: "NgNinja Academy says
hello world"
```

`apply`

- `apply` also attaches `this` to a function and invokes the function immediately
- `apply` is similar to `call()` except it takes an array of arguments instead of the comma-separated list

- In the below example `this` is set to the `ngNinja` object
- You can send arguments to the function as a comma-separated list following the owner object

```
var person = {
  hello: function(message) {
    console.log(this.name + " says hello " + message);
  }
}

var ngNinja = {
  name: "NgNinja Academy"
}

person.hello.apply(ngNinja, ["world"]); // output: "NgNinja Academy says
hello world"
```

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)
```

Promises

- Now that you have understood what is asynchronous programming and what are callbacks
- Let's dive into some advanced stuff - Promises
- Promises are basically another way to deal with asynchronous programming
- These simplifies your async code greatly!
- The example we saw earlier was contrived and simple - so you might not notice much difference
- BUT! in the real world applications promises simplifies the code to a great extent

Explanation via Example

- Let's implement the `fetchAndPrintUser` example using Promises

> TIP: When reading through this example try and compare with how we implemented the same requirement using callbacks

- As before we define the `fetchAndPrintUser` function which fetches the user details and prints the user
- But, this time instead of passing any callback function we create a new promise
- New promise can be created as below

```
const newPromise = new Promise()
```

```
What is a promise?

- Promise is literally a promise made by some function
- That it will eventually return the result and fulfill that promise
- Promise is a proxy for a value that will eventually become available
```

- The `Promise` object itself takes a callback function with two functions as parameters
- `resolve` - function to be called after successful data retrieval
- `reject` - function to be called if there was some error during data retrieval
- So, in the example below we return `Promise` from the `fetchAndPrintUser` function
- Once the data is available we return the data using `resolve(fakeUserName)`
- If there were any network error or some server failue - we would return error by rejecting the promise
    - This is done using `reject('Error ocurred!')`

```
function fetchAndPrintUser() {

  // create new promise
  return new Promise((resolve, reject) => {

    // simulate fake network call
    setTimeout(() => {

      // simulate error
      // when error occurs we reject the promise
      if(someError) {
        reject('Error ocurred!')
      }
      const fakeUserName = 'Sleepless Yogi'

      // Resolve the user name
      resolve(fakeUserName)
    }, 500)
  })

}
```

- The usage of promise is done via `promise.then.catch` pattern
- This means if the data is correctly resolved the execution goes in the `then()` block
  - Where you can do any other thing with the result data
- If the promise was rejected due to some error the execution would go in the `catch()` block
  - Where you can handle errors
- This is demonstrated below

```
// Execute function that fetch user and then prints it
fetchAndPrintUser()
  .then((name) => {
    console.log(name)
  })
  .catch((error) => {
    console.log(error)
  })
```

Promise.all

- Let's see how to handle if you want to fetch via multiple APIs and then perform some operation on the entire dataset
- This naive way would be to declare multiple promises and then perform operations when all promises are resolved

- Like below
- We create two different promises
- One for user data
- Another for order data

```
const userPromise = new Promise()

const orderPromise = new Promise()


// Wait for user data
userPromise.then((userData) => {

  // Wait for order data
  orderPromise.then((orderData) => {

    // after you get user and order data both
    // then perform some operation on both dataset
    console.log(userData, orderData)
  })
})
```

- Did you see how messy the code is
- If you had 3 or 10 or 100 promises - can you imagine how much nesting you would have to do?
- That is clearly bad!
- Enter `promise.all`!!!
- You can simplify the above code using `promise.all`
- Basically using this you can wait for all the promises to resolved and then only perform the next operations
- The above example can be written like below
- Please read the inline comments

```
const userPromise = new Promise()
const orderPromise = new Promise()

Promise.all([userPromise, orderPromise])
  .then((data) => {

    // here we are confident that we have both
    // user data as well as the order data
    console.log(data)
  })
  .catch((error) => {

    // we fall in this code block
    // if either one or all the promises are rejected
    console.log(error)
  })
```

Async-await

- Similar to callback and promises, we have another paradigm for handling async programming
- It is called Async-await
- This method is less verbose and much more readable
- If you are comfortable with synchronous programming this method will be much easy to understand
- Because it does not include callbacks

Explanation via Example

- For this to work we need two things
- One - `async` function
- Two - `await` on some promise
- If your function is awaiting on some asynchronous data you have to define your function as `async`
- And you have to use `await` keyword for the function call that is making the network API call

- Please see the example below
- We have defined `fetchAndPrintUser` function which fetches the user name and prints it
- Your function `fetchAndPrintUser` is defined as `async`
- Because internally it is calling `await fetchUserData()`
- `fetchUserData` is the function that is making network call to the API to fetch the user data

```
// Your async function
async function fetchAndPrintUser() {

  // await on the API call to return the data
  const name = await fetchUserData()

  // your data is now available
  console.log(name)
}
```

- Just see how simple and less-verbose the example looks
- You don't have to deal with callbacks or promises

Handle errors using async-await

- To handle errors using async-await you have to wrap the code inside `try-catch` block
- Like below

```
async function fetchAndPrintUser() {
  try {
    const name = await fetchUserData()

    // we have the data successfully
    console.log(name)

  } catch (error) {
    // there was some error
    console.log(error)
  }
}
```

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]
```

Arrow Functions

- They were introduced in ES6
- It is another syntax to create functions
- It has a shorter syntax

```
// syntax

(parameters) => { statements }
```

- Brackets around parameters are optional if you have only 1 param
- Statement brackets can be removed if you are returning an expression

- Below arrow function takes in `number` parameter
- It multiplies the number with 2
- And finally it returns the result

```
// example

var double = number => number * 2


// equivalent traditional function

var double = function(number) {
  return number * 2
}
```

Another example

- You can pass multiple parameters to the arrow function
- You can also write `{}` and return value like a normal function

```
// example

var sum = (a, b) => {
  return a + b
}

// equivalent traditional function

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

Lexical `this`

- It means forcing the `this` variable to always point to the object where it is physically located within
- This phenomenon is called as Lexical Scoping
- Arrow function let's you achieve a lexical `this` via lexical scoping
- Unlike a regular function, an arrow function does not bind `this`
- It preserves the original context
- It means that it uses `this` from the code that contains the Arrow Function

Example of lexical `this`

- Below example declares `person` object
- It has a `name: 'John'` and a function `printName()`

- When you invoke `printName()` using `person.printName()`
- The `this` operator originally points to the `person` object
- Therefore `this.name` logs `John` correctly

- Then we have declared two function `getName()` and `getNameArrowFunction()`
- Both of them does the same thing - they return the name of the person

- But, `getName()` gives an error because `this` is undefined inside the function
- Because in traditional function `this` represent the object that calls the function
- And we have not assigned any object to the function invocation

- Whereas, `getNameArrowFunction()` logs `John` correctly
- That is because it uses `this` object from the code that contains the Arrow Function which is `person`

```
var person = {
    name: 'John',
    printName: function(){

        console.log(this.name); // John

    var getName = function() {
      return this.name // ERROR
    }

    var getNameArrowFunction = () => {
      return this.name
    }

    // TypeError: Cannot read property 'name' of undefined
    console.log(getName())

    // John
    console.log(getNameArrowFunction())

    }

}

person.printName()
```

Destructuring Operator

- It lets you unpack values from arrays, or properties from objects, into distinct variables

Example using array

- You can name your variables anything

```
let [a, b] = [1, 2]

console.log(a) // 1
console.log(b) // 2
```

Example using object

- Your name of the variables should match the name of the properties
- Order does not matter

```
let { b, a } = {
  a: 1,
  b: 2
}

console.log(a) // 1
console.log(b) // 2
```

Rest Operator

- It allows us to more easily handle a variable number of function parameters
- Earlier we had to use `arguments` variable to achieve this

```
function log() {

  for(var i = 0; i < arguments.length; i++) {
    console.log(arguments[i])
  }
}

log(1) // 1
log(1, 2, 3) // 1, 2, 3
```

Using Rest Operator

- It will assign all the remaining parameters to a rest-variable after those that were already assigned
- `numbersToLog` is the rest-variable in the example below
- Rest operator puts all the remaining arguments in an array and assigns it to the rest-variable

> Rest operator turns comma-separated value to an array

```
function log(a, ...numbersToLog) {
  console.log(a) // 1
  console.log(numbersToLog) // [2, 3]
}

add(1, 2, 3)
```

Spread Operator

- It looks like has the same as the `Rest` parameter operator
- But it has a different use case
- In fact, it perform almost the opposite function to `Rest` operator

> Spread operator turns an array to comma-separated values

Example

- Below example spread `array1` to a comma-separated list of values into the `array2`

```
var array1 = [2, 3];
var array2 = [1, ...array1, 4, 5]; // spread

// array2 = [1, 2, 3, 4, 5]
```

Spread tricks

Concat array

```
const arr1 = ['coffee', 'tea', 'milk']
const arr2 = ['juice', 'smoothie']


// Without spread
var beverages = arr1.concat(arr2)

// With spread
var beverages = [...arr1, ...arr2]

// result
// ['coffee', 'tea', 'milk', 'juice', 'smoothie']
```

Make copy of array

```
const arr1 = ['coffee', 'tea', 'milk']
```

```
// Without spread
var arr1Copy = arr1.slice()


// With spread
const arr1Copy = [...arr1]
```

Remove duplicate entries from Array

```
const arr1 = ['coffee', 'tea', 'milk', 'coffee', 'milk']


// Without spread
// Iterate over the array add it to object as property
// If value present in the object skip it
// Else push it to another array


// With spread
const arr1Copy = [...new Set(arr1)]

// result
// ['coffee', 'tea', 'milk']
```

Convert string to array

```
const myBeverage = 'tea'

// Without spread
var bevArr = myBeverage.split('')

// With spread
var bevArr = [myBeverage]

// result
// ['t', 'e', 'a']
```

Find min max

```
// Without spread
var max = Math.max(3, 2, 1, 5, -10)


// With spread
var myNums = [3, 2, 1, 5, -10]
var max = Math.max(...myNums)

// result
// 5
```