JavaScript Core Concepts

By **Sunil Kumar**

Preface

This **JavaScript Core Concepts** handbook contains the core concepts of JavaScript.

The book does not try to cover every concept of JavaScript, instead it focuses on the minimum amount of knowledge required to work as an individual contributor.

This book is written by Sunil Kumar. My passion is to learn and deep dive in frontend development. I also write on topics like ReactJs, TypeScript, Angular, SCSS, NodeJs and various other frontend technologies and frameworks. I also publish UI related work on my website https://www.ui-geeks.in.

You can reach me at:
Email: skumar.mca2010@gmail.com
Twitter: @skumar_mca2010

Would appreciate your constructive feedback.

Enjoy reading the book

Table of Content

CHAPTER 1

Introduction

Preface to JavaScript

What is JavaScript

Lightweight, interpreted, JIT, prototype-based, multi-paradigm, OOPS

Interpreted or JIT Compiled

Interpreted with JIT Compilation in modern browsers

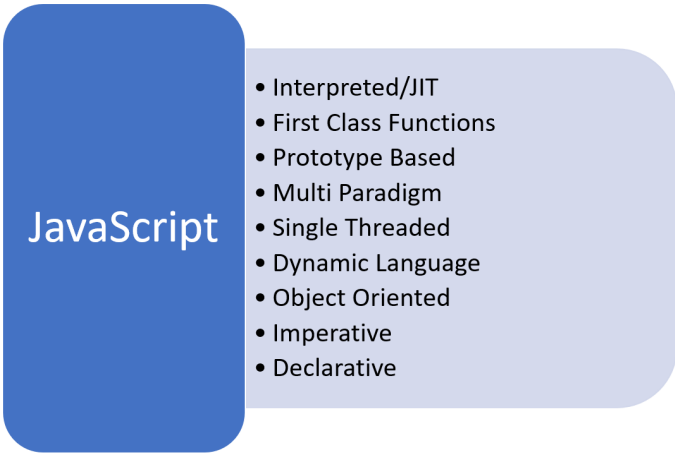
Features

First Class Function, Proto-type Based, Multi-paradigm, Single-Threaded, Dynamic Language, Object Oriented, Imperative and Declarative Styles

Introduction

JavaScript (JS) is a light-weight, **interpreted** or **just-in-time compiled** programming language with **first-class functions**. It is **prototype-based**, **multi-paradigm**, **single threaded**, **dynamic language**, supporting **object-oriented**, **imperative** and **declarative** styles.

Let's learn about all the buzz-words given in JavaScript definition.



Interpreted or Just-in-time Compiled

There has always been a confusion related to the execution behavior of JavaScript. The debate is always between Interpreted and Compiled. To know the answer, let's understand both terminologies.

Interpreted Language: In an interpreted language, the code is run from top to bottom and the result of running the code is immediately returned. The interpreter reads and executes one line at a time. The interpreter stops the execution on error, which means, if interpreter fails on a

statement at line number 10, further lines of code are not executed.

In the context of the browser, we don't have to transform the code into any other form before the browser runs it. The code is received in its plain-text format and processed directly from that.

Compiled Language: Compiled language scans the entire code and transforms (compiles) it into another form, before they are executed by the computer. The program is executed from a binary format, which was generated from the original source code (in plain-text format).

Just-in-time Compilation: Modern JavaScript interpreters use a technique called just-in-time compilation to improve performance. In this technique, JavaScript source code gets compiled into binary format, so that it can run quickly.

Now returning back to the original question, whether JavaScript is interpreted or compiled language?

The answer is that JavaScript is an **interpreted language**, because although modern interpreters use JIT, still all this compilation is handled at run time, rather than ahead of time.

First Class Functions

A programming language is said to have **first-class functions**, when functions are treated as other data types. For example: they can be stored in a variable, they can be passed as an argument to another function, can be extended, etc. JavaScript treats functions as first class citizens.

Prototype-based

Prototype based programming is a style of object oriented programming in which classes are not explicitly defined, rather derived by adding properties and methods to an empty object.

In simple words, this type of style treats an object as the ***prototype*** or the template for the creation of another object.

Multi-paradigm

JavaScript supports multiple programming paradigms like imperative, declarative, object-oriented, functional programming, therefore, JavaScript is said to be a multi-paradigm language.

Single Threaded

JavaScript has a single thread, which is used to execute the code. Because of this single thread, JavaScript is synchronous in nature. Although it has concepts like Call Stack, Memory Heap, Event-loop, which helps it perform asynchronous tasks.

Dynamic Language

Dynamic language is one, in which operations which are normally done at compile time, can be done at run time. Operations like Adding properties/methods to an object, changing class/object prototype.

Because JavaScript allows such operations, therefore, it is known as Dynamic Language.

Object-oriented

Object-oriented programming paradigm consists of classes/objects holding the data and respective methods that can be taken on the data.

Because JavaScript has the concept of class/objects and implements inheritance and various other concepts of object-oriented language, therefore it is known to support object-oriented programming style.

Imperative

Imperative Programming language is a language where instructions for computers are written in step-by-step manner. This explicitly describes the order of execution to achieve the end result.

*Imperative language describes "**How**" of the desired output.*

Because JavaScript statements can be written to describe the steps to get the desired result, therefore, it is known to be imperative in nature. For example, a for-loop can be written to iterate and print each item of an array.

Declarative

Declarative programming language is one, in which programs describe their desired results without explicitly listing steps that must be performed. The codes of such languages are very abstract in nature.

Functional and Logical programming languages are examples of declarative programming style.

*Declarative languages describe "**WHAT**" of the desired output.*

Because JavaScript implements functional programming style, therefore, it is known to be declarative in nature.

For example, the *sort()* method of *Array* can be called to sort all the items of the array, we do not need to write down the steps to sort the array.

History of JavaScript

While it is most well-known as the scripting language for web pages, it is also used in many non-browser environments like Node.Js, Apache CouchDB, Adobe Acrobat, etc.

It was created by Brenden Eich at NetScape in 1995. It was initially created to embed executable code in web pages, to make web pages interactive. But soon it became powerful enough to build an entire website using JavaScript.

JavaScript was originally going to be called "***LiveScript***", but it was renamed to "*JavaScript*" because of two reasons.

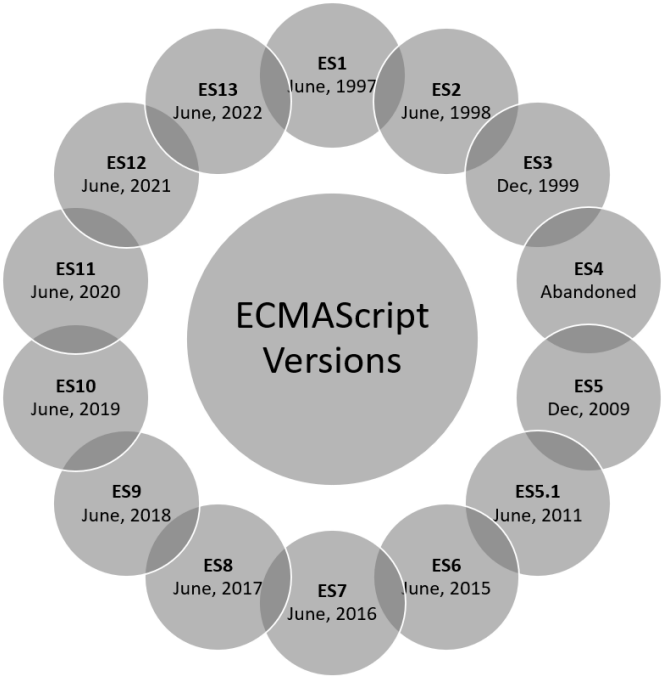
1. Many implementation ideas were taken from JAVA & C, like control structure, loops, etc.

2. An ill-fated marketing decision that attempted to capitalize on the popularity of JAVA.

Note: Both JAVA & JavaScript are registered trademarks of Oracle in the U.S. and other countries.

It was first released with Netscape 2 in 1996. Several months later Microsoft released JScript and after several months Netscape submitted JavaScript to Ecma

International, a European standards organization, which resulted in the first edition of the ECMAScript standard in 1997.



Since then many versions of ECMAScript have been released.

Version	Published Date
ES1	June, 1997
ES2	June, 1998
ES3	Dec, 1999
ES4	Abandoned, due to political differences concerning language complexity.
ES5	Dec, 2009
ES5.1	June, 2011
ES6	June, 2015
ES7	June, 2016
ES8	June, 2017
ES9	June, 2018
ES10	June, 2019
ES11	June, 2020
ES12	June, 2021
ES13	June, 2022

Note: Since ES6, a new version is released every year.

Summary

- JavaScript is an interpreted language.
- Modern JavaScript interpreters use JIT to improve performance.
- Functions can be treated as other types, they can be stored in variables, passed as function arguments, etc.
- JavaScript has a single thread, used for execution.
- Since JavaScript is Dynamic Language, properties can be added/deleted to objects, at run time.
- JavaScript supports Object-oriented, imperative and declarative styles.
- JavaScript is known as a scripting language for the web, but it also runs on non-browser environments.
- It was created by Brenden Eich in 1995, its first ECMAScript version was released in June, 1997.
- It was originally going to be called “LiveScript” but later renamed as “JavaScript”.

CHAPTER 2

Include & Execute JavaScript

Preface to Include & Execute JavaScript

How to include JavaScript Code in Webpage

Include Inline/External Scripts

Script Loading Strategies

How JavaScript is downloaded, parsed and executed

async

Download script without blocking page, execute after download, no guarantee of execution sequence

defer

Download script without blocking page, won't run until page content loaded, guarantee of execution sequence

Using JavaScript Console

Use Browser Console, console.log(), REPL

How to include JavaScript code in webpage

<script> tag is used for adding JavaScript code to a webpage. There are two ways of adding code via *<script>* tag.

1. **Inline Script**: In this, JavaScript code is written as the content of the *<script>* tag.

   ```
   <script>
       console.log("JavaScript code from inline script");
   </script>
   ```

2. **External Script**: In this, a file(.js), containing the code is linked to the webpage via the ***"src"*** attribute of the *<script>* tag.

   ```
   <script src="path-to-external-javascript-file"></script>
   ```

Note: If we write internal and external script in a single **<script>** tag, in this case, inline script will be ignored. For example:

```
<script src="path-to-external-javascript-file">
    console.log("I'll not be executed, will be ignored.");
</script>
```

Script Loading Strategies

All the HTML on a page is loaded in the order in which it appears. This behavior could create problems, when JavaScript is used to manipulate the DOM elements. Let's

discuss a problem statement, in order to understand it better.

Problem Statement:

If we use JavaScript to manipulate elements on the page, code will not work, if JavaScript is executed before the HTML is loaded.

Solution to the problem:

For Inline Scripts: The ***DOMContentLoaded*** event can be used to wait until all HTML elements are loaded.

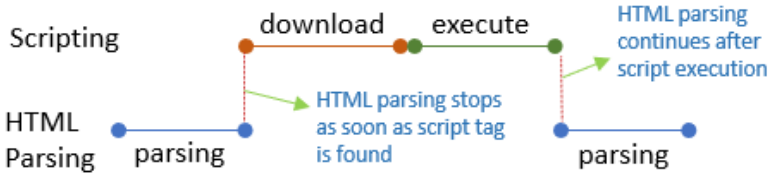
```
document.
  addEventListener('DOMContentLoaded', () => {
    // These statements will be executed
    // once all HTML is loaded
  });
```

For External Scripts: Before exploring the solutions, lets understand how external script affects the loading of the web page. Assuming our webpage just has HTML and external script files (not considering css, font, etc).

Browsers performs below activities:

- **HTML**
  Parses the HTML content to be rendered on the page.

- **JavaScript**
  Downloads the script file and executes the code. HTML parsing is stopped for this duration of download & execute, blocking the page for any user interaction. HTML parsing resumes once JavaScript has completed its execution.



Default script loading strategy

There are two ways to avoid blocking of page:

1. ***async*** attribute
2. ***defer*** attribute

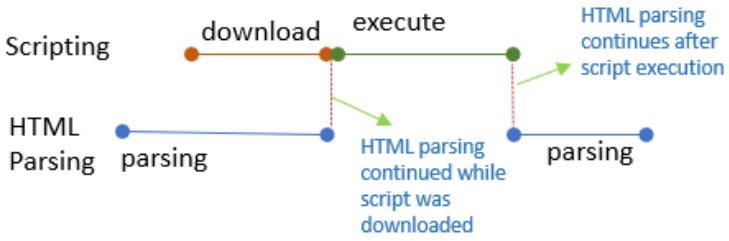
async

Scripts loaded using the ***async*** attribute on *<script>* tag will download the script without blocking the page. However, once the download is complete, the script will be executed, which blocks the page from rendering.

There's also no guarantee that scripts will run in any specific order. It is best to use *async* when the scripts in the page run independently.

*<script **async** src="script1.js" ></script>*
*<script **async** src="script2.js" ></script>*

Any of the two scripts can be downloaded/executed first.



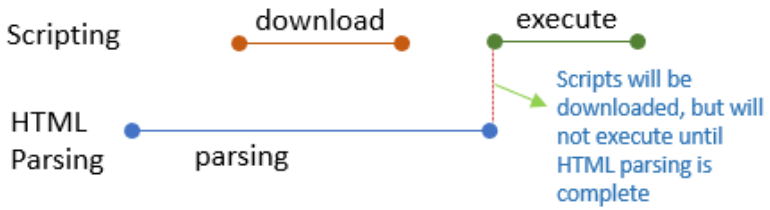
Script loading strategy with **async**

defer

Scripts loaded with the ***defer*** attribute will load in the order they appear on page. They won't run until page content has all loaded, which is useful if scripts depend either on the DOM or to each other.

*<script **defer** src="script1.js" ></script>*
*<script **defer** src="script2.js" ></script>*

Script1 will be downloaded/executed first.



Script loading strategy with **defer**

The ***async*** and ***defer*** both instruct the browser to download the scripts in a separate thread, while the rest of the page is loading, so the page loading is not blocked.

Note: An old-fashioned solution was to put all **<script>** tags at the **bottom of the <body>** tag, so that it would load after all HTML has been parsed.

The problem with this solution is that loading/parsing for the script is completely blocked until the HTML has been loaded. On larger sites with lots of JavaScript files, this can cause a **major performance issue**.

Using JavaScript Console for testing code

JavaScript could be tested in multiple ways. **'Browser Console'** being the most easily available, powerful yet simple tool. Every modern browser has a JavaScript Console, which allows users to write/test JavaScript code.

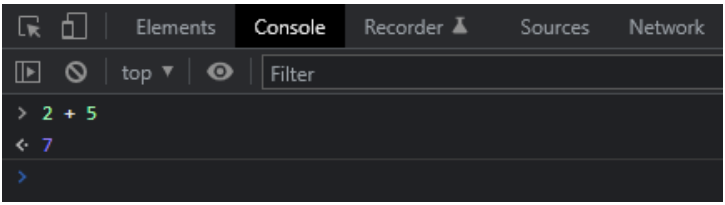
Ways of opening console: Below are the steps to open console in most popular browsers. Other browsers should also have a similar flow to the open console.

On Chrome/Safari/Opera browser

Right click on page > Click on **Inspect Element** > Click on **Console** tab.

On Firefox browser

Tools Menu > Click on **Web Developer** > Click on **Web Console**.

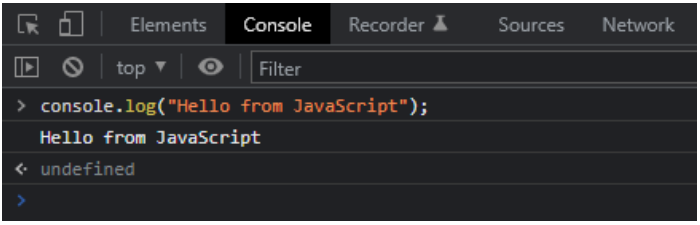


REPL (*Read-Eval-Print-Loop*) is running under the hood of every console.

REPL follows the given steps:

1. Read user input (i.e., JavaScript code expression).
2. Evaluate/Run the code.
3. Print the result.
4. Loop back to Step 1 (i.e., wait for user input).

Try running statement: ***console.log('Hello from JavaScript");***



The above statement will give two lines of output. The first line will print "'*Hello from JavaScript*" (as the output of the statement). The second line will print "*undefined*" (as the result of evaluating the expression).

In JavaScript, every expression has a result. But few expressions like *console.log()* returns nothing and in JavaScript ***"nothing"*** is represented as ***"undefined"***.

Summary

- <script> tags is used for adding JavaScript code to webpage.
- HTML on the page is loaded in the order in which they appear.
- In **"Default Script Loading"** strategy, HTML parsing is paused during the period of script download and execution.
- With the **async** attribute, the script is downloaded without blocking the page.
- Scripts with **async** attributes can be downloaded/executed in any order.
- With the **defer** attribute, scripts will not be executed, until everything is loaded.
- Scripts with **defer** attributes will execute in the given order.
- Browser Console can be used to test JavaScript code.
- REPL is running under the hood of every console.
- In JavaScript, all expressions return a result, few like *console.log()* return ***undefined***.

CHAPTER 3

types

Preface to types

Building blocks of JavaScript: the types

Number, BigInt, String, Boolean, Function, Object, Symbol, undefined, null, Array, Date, RegExp

Number

64-bit double precision, store number & floating, octal, hexa-decimal, parseInt(), parseFloat(), isNaN()

String

Unicode characters, 16-bit, immutable, toUpper(), charAt(), etc

BigInt

Number with arbitrary precision, BigInt(), append n

Symbol

Unique, immutable primitive value, Symbol(), Symbol.for(), Symbol.keyFor()

null

Deliberate non-value, behaves as zero (0)

undefined

Uninitialized value, behaves as NaN/false

Boolean

Possible values true and false, logical AND (&&), OR (||) and NOT (!)

Building block of JavaScript: the *types*

Primarily there are **7** primitive types in JavaScript

- Number, BigInt, String, Boolean, Function, Object, Symbol

But there are couple of other types too available in JavaScript

- undefined, null, Array, Date, RegExp

Also *functions* are just a special type of *object* (because functions are treated as first-class objects).

On Summarizing all, the type diagram will look something like:

- Number
- BigInt
- String
- Boolean
- Symbol (New in ES 2015)
- Object
    - Function
    - Array
    - Date
    - RegExp
- null
- undefined

> **Note**: There are some built-in **Error** types as well.

Number

The Number type is a double-precision-64-bit binary format IEEE 754 type. It can store numbers between -(2^53 - 1) to 2^53.

It can store both integer and floating point numbers, which may result in some strange behavior. For example:

```
console.log(3/2);   // it will log 1.5 and not 1
console.log(0.1+0.2);   // it will log 0.30000000000000004
```

The standard arithmetic operations are supported like addition, subtraction, modulus (remainder), etc.

```
10 + 50;  // 60
2 * 5;    //10
50 % 4;   // 2
```

There is also a built-in object ***"Math"*** for performing mathematical operations like *Math.sin()*, *Math.pow()*, etc.

```
Math.sin(3.5); // -0.35078322768961984
Math.PI; // 3.141592653589793

Math.pow(2, 3); // 8
//(Equivalent to 2 raised to the power 3)
```

parseInt()

This *function* is used to convert string value to integer.

Syntax
*parseInt("String representation of Number", **base**)*

base: It is opt*ional,* default value is **10.**

```
1  parseInt('123');      // 123
2  parseInt('123', 10);  // 123
```

In older browsers, strings beginning with a "0" (**Zero**) were assumed to be in **OCTAL** base (*radix 8*).

```
1  parseInt('010'); // 8 (Octal value of 010)
```

But in modern browsers, the above statement will print 10.

```
1  parseInt('010'); // 10
2  //(Not treated as OCTAL, used default base 10)
```

Hexa-decimal values are still treated the same. Strings beginning with "**0x**" or "**0X**" are treated as Hexa-decimal numbers (*radix 16*).

```
1  parseInt('0x10'); // 16 (Hexa-decimal value of 0x10)
```

Therefore, it's always safer to pass '***base***' while working on Octal and Hexa-decimal values.

```
1  parseInt('010', 8); // Will always give 8
2  parseInt('0x10', 16); // Will always give 16
```

parseFloat()

This *function* is used to convert *strings* to floating point numbers. It always uses ***base 10***.

Syntax
parseInt("String representation of Number")

```
parseFloat('1.23'); // 1.23
```

Unary operator "+" can also be used to convert strings to numbers.

```
+'42'; // 42
+'010'; // 10
+'0x10'; // 16
```

A special value called ***NaN*** ("Not a Number") is returned if the string is non-numeric.

```
parseInt('hello'); // NaN
```

Note: **NaN** is toxic, if it is provided as operand to any mathematical operation, the result will also be NaN.

For example:
*NaN + 5 // **NaN***

Number.isNaN()

This *function* can be used to check if a given value is *NaN*.

```
Number.isNaN('hello');  // false
Number.isNaN(NaN);      // true
```

Do NOT use the global *isNaN()* function for checking *NaN* values, since it has unintuitive behavior.

```
isNaN('hello')    // true
isNaN(undefined)  // true
isNaN([1])        // false
isNaN('1')        // false
isNaN({})         // true
```

Javascript also has special values ***Infinity*** and ***-Infinity***.

```
1 / 0;   // Infinity
1 / 0;   // -Infinity
```

isFinite()

This *function* can be used to check whether value is finite or not.

```
isFinite(100)   // true
isFinite(-Infinity) // false
isFinite(1/0)   // false
```

Note: ***parseInt()*** and ***parseFloat()*** functions parse a string until they reach a character that isn't valid for the specified number format, then returns the number parsed up to that point. However, the "+" operator converts the string to ***NaN***, if there is any invalid character contained within it.

parseInt("12.25abc") // **12**
parseFloat("12.25abc") // **12.25**
+ "12.25abc" // **NaN**

String

Strings in JavaScript are sequences of **unicode** characters. More accurately, they are sequences of **UTF-16** code units. Each code unit is represented by a 16-bit number. Each unicode character is represented by either 1 or 2 code units. Strings are **immutable** and cannot be modified.

Properties/methods of String:

```
'hello'.length; // 5
'hello'.charAt(1); // 'e'

'hello, world '.replace('world', 'Javascript');
// 'hello, JavaScript'

'hello'.toUpperCase(); // 'HELLO'

'hello'.substr(1, 2);
// 'el' (Returns 2 characters, starting at index 1)

'hello'.substring(1, 3);
// 'el' (Returns characters between given indexes)
```

BigInt

It can represent integers with arbitrary precision. With BigInts we can store and manipulate large integers, even beyond the safe Integer limit for Numbers.

A *BigInt* is created by appending **'n'** to the end of an integer or by calling the ***constructor***.

12345699924n
BigInt(12345699924)

> **Note**: *BigInt* cannot be operated on interchangeably with the Number. A TypeError will be thrown on such operations.

Symbol

A symbol is a unique and immutable primitive value and may be used as the key of an object property. Symbol value is always unique.

```
1  const obj = { [Symbol('abc')]: 'hello' };
2
3  Symbol('abc') === Symbol('abc'); // false
4  // (Because Symbol() will always populate a unique key)
```

Symbol.for()

In contrast to *Symbol()*, which always creates a unique key every time, *Symbol.for()* will create a new unique, if it is not yet created, otherwise, it will read and return an existing symbol with the given key from the "**global Symbol registry**".

```
Symbol.for('abc'); // Created a unique key

Symbol.for('abc');
// Returned from global symbol registry
```

Symbol.keyFor()

This *function* returns a key for the given Symbol.

```
const sym = Symbol.for('abc');
Symbol.keyFor(sym); // 'abc'
```

Note: Symbol.keyFor() do not work with Symbol()

null

It is a type that indicates a deliberate non-value and it is only accessible through the ***null*** keyword.

let t = null;

The null value behaves as **0 (Zero)** in numeric context and as ***false*** in boolean context.

```
null * 10; // 0
null && true; // false
```

undefined

It is a type that indicates an uninitialized variable value, i.e., a value hasn't been assigned yet to the variable.

```
let t;
console.log(t); // 'undefined'
```

The *undefined* value behaves as ***NaN*** in numeric context and as ***false*** in boolean content.

```
undefined + 10; // NaN
undefined && true;  // false
```

Boolean

JavaScript has a boolean type, with possible values of **true** and **false**. Many built-in operations that expect booleans first coerce their argument to booleans, for example, *undefined* and *null* are coerced to *false* while all *objects* are coerced to *true*.

- *false*, *0*, empty string (""), *NaN*, *null*, *undefined* becomes ***false***.
- All other values become ***true***.

```
Boolean(''); // false
Boolean(23); // true
```

Boolean operations like **&&** (Logical AND), **||**(Logical OR), **!** (Logical NOT) are supported.

```
Boolean(1) && Boolean(false)  // false
Boolean(1) || Boolean(false)  // true
!Boolean(1) // false
```

Note: We can use the ***Boolean()*** function to check if an expression/variable is *true*.

Summary

- There are **7** primary types in JavaScript.
- ***Number*** type can store both integer and floating point numbers.
- Unary operator **"+"** can also be used to convert string to number.
- ***NaN*** is toxic, operations with NaN will always return NaN.
- ***BigInt*** represents integers with arbitrary precision.
- ***Symbol()*** always returns a unique value.
- *false*, *0*, *empty string ("")*, *NaN*, *null* and *undefined* are treated as **false**.

CHAPTER 4

Grammar

Preface to Grammar

Introduction

JavaScript is case-sensitive, unicode statements separated by semicolon

Comments

Single/Multi line, HashBang Comments

Variables

Identifiers, let, const, var, Scope, Naming rules

Hoisting & Type Conversion

Lift variables to top, Dynamically types implicit and explicit type conversion

Literals

Represents valid values, Number, String, Boolean, Array, Object and RegExp Literals, String Interpolation

Grammar and types

JavaScript is ***case-sensitive*** and uses the ***Unicode*** characters set. In JavaScript, instructions are called ***statements*** and are separated by *semicolon* (**;**). A semicolon is not necessary after a statement, if it is written on its own line. But if more than one statement is written in a line, then they must be separated by semicolons.

Note: It is considered best practice to always write a semicolon after a statement. This practice reduces the chances of errors in the code.

Note: ECMAScript also has rules of Automatic Semicolon Insertion(ASI). Some JavaScript statements must be terminated with semicolon and are therefore affected by ASI. Few cases where ASI is executed:

- Variable declaration using **let**, **const**, **var**.
- Module declaration using **import**, **export**.
- Expression statement.
- **debugger**, **continue**, **break**, **throw**, **return**.

The source code of JavaScript is scanned from ***left*** to ***right***, and is converted into a *sequence of input elements* which are ***tokens***, ***control characters***, ***line terminators***, ***comments*** or ***whitespaces*** (*Spaces*, *tabs* and *new line characters* are considered as whitespace).

Comments

Comments should be added to improve code readability by explaining the purpose of code. Comments behave like whitespaces and are discarded during script execution.

Single Line Comment

Single line comments start with **//** character. As the name suggests, comments can't be spread in multiple lines, it has to be contained within a single line.

```
1  // I am a single line comment
2  // There can be multiple single line comments
```

Multi Line Comment

Sometimes it is required to write a more elaborated explanation, which spreads in multi line, multi line comment could be used for these cases.

Multi line comment start with **/*** and should end with ***/** characters.

```
1  /*
2  I am a multi line comment,
3  and can be use to write an
4  elaborated comment.
5  */
```

Note: There's a third type of comment syntax written at the start of some JavaScript files, which looks like **#!/usr/bin/env node**. This is called **HashBang Comment** syntax, and is a special comment used to specify the path to a particular JavaScript engine that should execute the script.

Variables

Variables are used as symbolic names for values. The names of variables are called ***identifiers***. New variables in JavaScript are declared using one of the below keywords:

- let
- const
- var

let

This allows us to declare ***block-level*** variables. The declared variable is available from the ***block*** it is enclosed within. In JavaScript, a ***block*** is the body within { } braces.

Syntax

let *<variable-name>=<optional-assignment>;*

```
let name = 'JavaScript';
let unInitializedVariable;
```

Scope of let

```
// letVar is not visible here
for (let letVar = 1; letVar < 5; letVar++) {
  // letVar is visible here,
  // it is only available within the block
  // it is enclosed within.
  // Here block is the for loop.
}
// letVar is not visible here
```

const

This is also used to declare ***block-scoped*** variables but whose values are never intended to change through reassignment or re-declaration. It is also available from the block it is declared within.

A *const* variable should be initialized. We must specify its value while declaring it.

Syntax

const *<variable-name>=<assignment–constant-value>;*

```
const PI = 3.14;

PI = 4;
// This will throw an error because
// we cannot change a constant variable.
```

Note: Global constants do not become properties of the *globalThis* object, unlike *var* variables.

Properties of **Object** and **elements of an Array** are *not protected*, even declaring with ***const*** keyword.

```
const obj = { a: 10 };
obj.a = 20;
console.log(obj); // { a : 20 }

const arr = [1, 2, 5];
arr.push(10);
console.log(arr); // [1, 2, 5, 10]
```

var

It was the *traditional* (older) way of declaring variables, therefore, it doesn't have restrictions of block-level.

Variables declared with the ***var*** keyword have ***function/global level scope*** and therefore are accessible through the *function* it is declared within.

Variables declared using the *var* are created before any code is executed, this process is called **hoisting**.

Syntax
var *<variable-name>=<optional-assignment>;*

```
// uninitialized variable,
// default value is undefined.
var name;

var name = 'JavaScript';
```

In the global context, the *var* variable is added as a non-configurable property in the global (***this***/***globalThis***) object.

Note: Duplicate variable declaration using *var* will not trigger error, even in *strict-mode*.

Scope of var

```
// myVar is visible here
for (var myVar = 1; myVar < 5; myVar++) {
  // myVar is visible here
}
// myVar is visible here
// It is available outside the block
// because it has function/gloal level scope.
```

Variable naming rules

Generally, we should stick to just using *Latin characters* (**0-9, a-z, A-Z**) and the **underscore** character. There are some rules/conventions to be followed for naming a variable instance.

- Prefer **camelCase** naming convention.
  For example: *projectName*, *userPhone*, etc.
- Prefer using **underscore**(_) for defining multi word names.
  For example: *customer_Address*, *inventory_management_module*, etc.
- ***Don't use underscore*** (_) at the *start of the name*, because in certain JavaScript constructs, it is used to declare global variables.
  For example: *_globarUser*.
- Make variable names **intuitive**, so they describe the data they hold.
- Variable names are **case sensitive**.
  For example *myVar* is different from *myvar*.

- Variable names **starting with numbers are not allowed**.
  For example: *9varName* would be an invalid name.

- **Reserved** keywords should **not** be used.
  For example: *for* would be an invalid name.

Evaluating Variables

Below given rules are applied while evaluating variables.

- Variable declared using the ***var*/*let*/*const*** with no assignment, has the value of ***undefined***.

- Attempting to access an undeclared variable results in **ReferenceError**.

- The ***undefined*** value behaves as ***false*** in *boolean* context and ***NaN*** in numeric context.

```
undefined + 10;       // NaN
undefined && true;    // false
```

- The ***null*** value behaves as ***false*** in *boolean* context and **0 (Zero)** in numeric context.

```
null + 10;            // 10
null && true;         // false
```

Variable Scope

When a variable is declared *outside of function*, it is called a **global variable**, because it is available to any other code in the current document.

When a variable is declared *inside a function*, it is called a **local variable**, because it is available only within that

function. ***let*** and ***const*** declarations are ***scoped to the block*** that they are declared within.

```
let globalVariable = 10;
anotherGlobalVariable = 20;

function demoVariableScope() {
  let localVariable = 50;
  for (let blockVar = 1; blockVar < 5; blockVar++) {
    console.log(blockVariable);
  }
  console.log(blockVariable); // Not available here
}

console.log(globalVariable); // Available here
console.log(localVariable); // Not available here
```

Variable Hoisting

In JavaScript, we can refer to a variable which is declared later, without getting an exception/error. This concept is known as **hoisting**.

Variables in JavaScript are **"hoisted"** (or **"lifted"**) to the top of the function or statement. However, variables that are hoisted return a value of ***undefined***.

```
console.log(x === undefined); // true
var x = 10;

var a = 100;
(function () {
  console.log(a); // undefined
  var a = 50;
})();
```

let and ***const*** are hoisted but not initialized. Referencing the variable in the block before the declaration results in ***ReferenceError***, because the variable is in a "**temporal dead zone**" from the start of the block until the declaration is processed.

```
console.log(x); // ReferenceError
let x = 10;
```

Temporal Dead Zone (TDZ)

A variable declared with ***let*** or ***const*** is said to be in **"temporal dead zone" or TDZ**, from the start of the block until code execution reaches the line where the variable is declared and initialized.

While inside the TDZ, the variable has not been initialized with a value and an attempt to access it will result in **ReferenceError**. This differs from ***var*** variables, which will return a value of ***undefined***, if they are accessed before they are declared.

```
{
  // TDZ starts at the beginning of the scope
  console.log(varVariable); // undefined
  console.log(typeof varVariable); // undefined
  console.log(letVariable); // ReferenceError
  console.log(typeof letVariable); // ReferenceError

  var varVariable = 10;
  let letVariable = 20;
  // TDZ ends at the end of the scope
}
```

The term **"temporal"** is used because the zone depends on the order of *execution (time)*, rather than the order in which the code is *written (position)*.

Implicit Type Conversion

JavaScript is a ***dynamically typed*** language, which means we don't have to specify the *datatype* of a variable on declaration.

The data types are automatically converted *as-needed* during the script execution, this conversion is known as **implicit** conversion.

```
1  let a = 10;
2  a = 'JavaScript';
```

*The type of **"a"** is automatically converted from **number to string**.*

Literals

Literals represent fixed ***values*** in JavaScript. These are the values of the variables and not the type of the variables. Literals are often used to initialize variables.

Types of *literals*:

- Numeric Literals
- String Literals
- Boolean Literals
- Array Literals
- Object Literals
- RegExp Literals

Numeric Literals

Any valid integer and floating point numbers. This includes integer values in different *radix bases* as well as floating-point literals in **base-10**.

```
let a = 20;
let b = 12.25;
let c = 011;
let d = 0x1123;
```

String Literals

A string literal is zero or more characters enclosed in single (') or double (") quotes. All the ***String*** object methods can be called on a string literal.

JavaScript automatically converts the *string literal* to a temporary ***String*** *object*.

```
const str = 'JavaScript Literal';
console.log('JavaScript'.length); // 10
```

length *is a property of the String object, which can be called on string literal*.

Template Literals

These are string values enclosed by the ***back-tick*** **(`)** character instead of double or single quotes. Also known as **string interpolation**. They provide syntactic sugar for constructing strings. The **${}** construct is used to add dynamic/variable value to the string literal.

```
// Multiline string values can be created.
const multiLineStr = `I am a multiline
string`;

// Add dynamic value to string literal
const name = 'UI Geeks';
const str = `My name is: ${name}.`;
console.log(str); // 'My name is: UI Geeks.'
```

Tagged Templates

These are a compact syntax for specifying a template literal along with a call to a **"tag" function** for **parsing**.

A tagged template is just a more succinct and semantic way to invoke a function that processes a string with a set of relevant values.

```
const formatArgument = (arg) => {
  if (Array.isArray(arg)) {
    return arg.map((itm) => `# ${itm}`).join('\n');
  }

  if (typeof arg === 'object') {
    return JSON.stringify(arg);
  }

  return arg;
};

const printStr = (segments, ...args) => {
  let msg = segments[0];
  segments.slice(1).forEach((segment, index) => {
    msg += formatArgument(args[index]) + segment;
  });
  console.log(msg);
};
```

segments: This holds the list of ***fixed text*** of the string literal.

Args: Holds the list of ***dynamic values*** of the string literal.

For any valid template literal, there will always be **N *args*** and ***(N+1)*** string ***segments***.

```
const todos = ['JavaScript', 'ReactJs'];
const progress = { js: 90, react: 60 };

printStr`I am learning:\n${todos}
\nAnd my progress is: ${progress}`;

// Output:
// I am learning:
// # JavaScript
// # ReactJs
// And my progress is: { 'js' : 90, 'react': 60 }
```

Note: **console.log()** uses similar string interpolation.
*console.log("Learning **%o** with **%o** percent progress", "JS", 50);*
// "Learning 'JS' with 50 percent progress"

Boolean Literals

Boolean type has only 2 literal values: ***true*** and ***false***.

Array Literal

An array literal is a list of zero or more expressions enclosed in square brackets (**[]**).

const arr = [1, 2, 10, 20];

If we put two commas in a row in an array literal, the array **leaves the empty slot** for the unspecified elements. Array traversal methods like *map()*, skip the empty slots.

```
1  const arr = [1, 2, 10, , 20];
2  // [1, 2, 10, empty, 20]
```

If we put a trailing comma at the end of the list, then the comma is ignored.

```
1  const arr = [1, 2, 10,];
2  // [1, 2, 10]
```

Object Literal

An object literal is a list of zero or more pairs of property name and value, enclosed in **{}**.

```
1  const obj = { a: 10, 'invalid-key': 20 };
```

Property names that are **valid identifiers**, can be accessed with **dot notation**.

```
1  const obj = { a: 10, 'invalid-key': 20 };
2  console.log(obj.a); // 10
```

Property names that are **not valid identifiers**, can be accessed with **bracket notation**.

```
const obj = { a: 10, 'invalid-key': 20 };
console.log(obj['invalid-key']); // 20
```

RegExp Literals

A regex literal is a pattern enclosed between backward slashes (**//**).

```
const re = /ab+c/;
```

Summary

- JavaScript is **case sensitive**, instructions are called **statements** which are separated by **semicolon**.
- **Single/Multi-line *comments*** should be added to improve code readability.
- **Variables** are symbolic names for values and their names are called **identifiers**.
- ***let*** and ***const*** declare **block scope** while ***var*** declare **function/global scope** variables.
- **Variable** names should include *Latin* characters **(0-9,a-z,A-Z)** and **underscore** "_".
- Prefer naming variables using **camelCase**, **intuitive** names.
- Unassigned variables will return ***undefined***.
- Variables can be declared in **global**, **local** and **block scope**.
- Variables are **"hoisted"** to top of function, so they can be referred before their declaration.
- JavaScript is **dynamically typed**, so *data types* are converted **as-needed** during execution.
- **Literals** represent values in JavaScript.
- **Template Literals** are string values enclosed by the **back-tick(`)**.
- *Trailing commas* at the end of **Array literal** are **ignored**, while **two commas** in a row adds an ***empty*** slot.
- Properties of Object can be read by **dot (.)** or **bracket ([])** notation.
- The ***let*** and ***const*** variables are in the **Temporal Dead Zone (TDZ)** from *start of scope* to their *declaration* statement.
- Accessing variables of the **Temporal Dead Zone** will give **ReferenceError**.

CHAPTER 5

Operators

Preface to Operators

Introduction
Unary, binary and ternary operators

Unary Operators
Prefix, Postfix, NOT, +, delete, typeOf, void, in and instanceOf Operators

Binary Operators
Arithmetic, Relational, Equality, Assignment, Logical and Bitwise operators, Short-circuit Evaluation

Ternary Operators
? : Operator

Operator Precedence
Priority and direction of evaluation

Operators

JavaScript has a rich set of operators, which can be used to perform operations on given values. JavaScript has both ***binary*** and ***unary*** operators, and one special ***ternary*** operator.

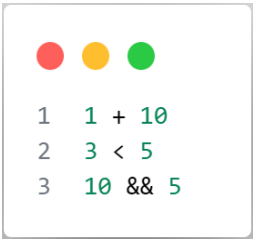
Binary Operators

A binary operator requires **two operands**, one before and one after. This form of expression is called an ***infix*** binary operator, because the operator is placed in-between two operands.

All binary operators in JavaScript are infix.

Syntax

operand1 ***operator*** *operator2*



```
1  1 + 10
2  3 < 5
3  10 && 5
```

Binary operators can be divided in given categories:

- **Arithmetic** operators (+, -, *, /, %, **)
- **Relational/Comparison** operators (<, >, <=, >=)
- **Equality** operators (==, !=, ===, !==)
- **Assignment** operators (=, +=, -=, *=, /=, %=)
- **Binary Logical** operators (&&, ||)
- **Binary Bitwise** operators (&, |, ^)

Arithmetic Operators

These are normal mathematical operators, which we have learned in school.

```
10 + 50;  // 60
5 * 4;    // 20

20 % 6; // 2
// (Modulus operator, returns the remainder.

2 ** 3;   // 8
// (Exponential operator)
```

Relational Operators

Used for comparing two values. It compares the two operands and returns a boolean value, based on whether comparison is ***true*** or ***false***.

Operator		Example
<	Less than	2 < 5
>	Greater than	5 > 2
<=	Less than equal to	2 <= 2
>=	Greater than equal to	2 >= 2

Equality Operators

Used to check the equality of the operands, and returns a boolean value, based on whether comparison is ***true*** or ***false***.

Type Coercion

The **double equality** (==) and **inequality** (!=) operator performs type coercion, if it receives different types.

Type Coercion is automatic or implicit conversion of values from one data type to another.

==	Attempts to convert (*Type Coercion*) and compare operands for equality. Returns *boolean* value.
	
	*"1" == 1; // **true***
	*"hello" == "hello"; // **true***

!=	Attempts to convert (*Type Coercion*) and compare operands for inequality. Returns *boolean* value.
	
	*"1" != 1; // **false***
	*"Hello" != "hello"; // **false***

===	Strict equality operator, checks for equality of operands, without converting (*Type Coercion*) the values before comparison.
	
	This considers operands of different types to be different.
	
	*"1" === 1; // **false***
	*1 === 1; // **true***

!==	Strict inequality operator, checks for inequality of operands, without converting (*Type Coercion*) the values before comparison.
	
	This considers operands of different types to be different.
	
	*"1" !== 1; // **true***
	*2 !== 2; // **false***

Assignment operators

Assignment operator assigns a value to *left operand* based on the value of the *right operand*, in short, it assigns right hand operand value to left hand side operand.

let x = 10;

There are also ***compound assignment*** operators that are shorthand for linear assignment operators.

x += 10;
This is equivalent to, ***x = x + 10;***

Similarly we have other compound assignment operators, **-=**, ***=**, **/=**, **%=**, ****=**

Binary Logical Operators

Logical operators are typically used with ***boolean values*** and return a boolean value. However, if they are used with non-boolean values, they may return non-boolean values.

Logical AND (&&) operator
operand1 ***&&*** *operand2*

When used with **boolean values**, returns ***true*** if both operands are *true*, otherwise returns **false**.

```
1  false && true; // false
2  true && true; // true
```

When used with **non-boolean** values, returns *operand1* if it can be converted to *false*, otherwise returns *operand2*.

```
'JS' && 'Program';
// 'Program' (Because first operand 'JS'
// cannot be converted to false)

false && 'Program';
// false (Because first operand is false)

'Program' && false;
// false (Because first operand 'Program'
// can't be converted to false,
// hence the second operand is returned.
```

Logical OR (||) operator

operand1 **||** *operand2*

When used with **boolean values**, returns **true**, if any operand is *true*, otherwise returns **false**.

true || false; *// **true***

When used with **non-boolean values**, returns ***operand1***, if it can be converted to ***true***, otherwise returns ***operand2***.

```
'JS' || 'Program';  // 'JS'

false || 'Program'; // 'Program'
// (Because operand1 can't be converted
// to true, therefore, returned operand2)
```

Short-circuit evaluation

Because logical operators are evaluated from left to right, they are tested for possible **"short-circuit"** evaluation using below rules.

Rule 1: *false* && ***expression***, is short-circuit evaluated to ***false***, therefore, *expression* will not be evaluated/executed.

```
evt && evt.target && evt.target.value;
```

evt **expression1**	*evt.target* **expression 2**	*evt.target.value* **expression3**

expression2 *will not be evaluated until* ***expression1*** *results* ***true*** *and* ***expression3*** *will not be evaluated until* ***expression2*** *results* ***true***.

```
callback && callback();
// callback() will be executed if it's
// defined i.e., callback as expression1
// results true.
```

Rule 2: ***true*** || *expression*, is short-circuit evaluated to ***true***.

```
const msg = null;
const res = msg || 'Default Message';
```

msg **expression1**	*"Default Message"* **expression2**

"Default Message" is assigned to the res *variable*, because ***expression1*** is not *true*, therefore ***expression2*** is returned.

```
1  const res = 0 || 'Hello';
```

"Hello" is assigned to the *res variable*, because the Logical OR operator treats *0, "", NaN, null, undefined* as **falsy** value, therefore 0 (zero) was not returned.

But what if we want 0 (zero) to be returned, basically to treat only null and undefined as falsy value. We can use the **Nullish Coalescing Operator (??)**.

```
1  res = 0 ?? 'Hello';   // 0
2  res = null ?? 'Hello';    // 'Hello'
3  res = undefined ?? 'Hello'; // 'Hello'
```

Unary operators

A unary operator requires a single operand, either before or after the operator.

Syntax

operator *operand*

This form is called ***prefix*** unary operator.

```
1  let x = 10;
2  ++x; // 11
```

Logical NOT (!) Operator

Returns ***false*** in *expression* can be converted to *true*, otherwise return ***true***. In short, it **inverses** the *truthy/falsy* value, i.e., inverses *true* to *false* and vise-versa.

Syntax

!expression

```
!true; // false
!false; // true
!'JS'; // false
```

Double NOT (!!) Operator

It is used to explicitly force the conversion of any value to the corresponding *boolean primitive*.

```
!!true; // true
!!{}; // true
!!""; // false
```

Unary operator "+"

This operator can also be used to convert strings to numbers.

```
+'42'; // 42
+'010'; // 10
+'0x10'; // 16
```

The **+** operator can also be used for string concatenation.

```
'Hello' + 'JavaScript'; // 'Hello JavaScript'
```

Note: If we add a string to a number (or other value), everything is converted into string first.

"3" + 4 + 5; *// "345"*

3 + 4 + "5"; *// "75"*

Adding an empty string to something is a useful way of converting it to string.

delete Operator

The *delete* operator deletes an object's property, but can't delete the entire object. Returns ***true***, if delete operation was successful, otherwise returns ***false***.

```
const obj = { name: 'JavaScript', version: 6 };
delete obj.version; //  true
console.log(obj.version); // undefined
```

The *delete* operator can't delete *non-configurable* or *system defined* properties.

```
delete Math.PI; // false
```

Deleting array elements

The ***delete*** operator should **not** be used for deleting array elements. It is possible to delete any array elements, although it is regarded as **bad practice**, because it does not actually delete the element (it just sets the element as *undefined*) and therefore, length property is not re-calculated.

```
const arr = [1, 5, 10, 20, 40];
delete arr[2]; // true

console.log(arr);
// [1, 5, undefined, 20, 40]
```

To manipulate arrays, array methods like splice() should be used.

typeof Operator

This returns a *string* indicating the type of the operand. Parentheses around *typeof* are optional.

Syntax

typeof *operand* **OR** ***typeof****(operand)*

```
const show = () => { console.log('Hello'); };
typeof show;  // 'function'

const name= 'JavaScript';
typeof name;  // 'string'

const arr = [1, 5];
typeof(arr);  // 'object'
```

void Operator

This operator specifies an expression to be evaluated without returning a value.

Syntax

void *expression* **OR** ***void****(expression)*

```
1  const res = void 3 + 5;
2  console.log(res); // undefined
```

in Operator

The ***in*** operator returns **true**, if given property is in the specified *object*.

Syntax

propertyName ***in*** *object*

```
1  const obj = { name: 'JavaScript', version: 6 };
2  console.log(name in obj); // true
3  console.log(arr in obj); // false
```

The ***in*** operator also works with **Arrays**, but it returns ***true***, if a given **index** is found in the *array*.

```
1  const arr = ['A', 'B', 'C'];
2  console.log(0 in arr); // true
3  console.log(5 in arr); // false
4  console.log('B' in arr); // false
```

instanceof Operator

The *instanceof* operator returns ***true***, if the given *object* is of the given object type.

Syntax
objectName ***instanceof*** *objectType*

```
const day = new Date();
console.log(day instanceof Date); // true
```

Ternary (?:) Operator

It is the only JavaScript operator that takes **three operands**.

Syntax
operand1 ? operand2 : operand3

Better written as:
condition ? operand1 : operand2

If the condition is *true*, **operand1** is returned , otherwise **operand2** is returned.

```
let num = 10;
const msg = num % 2 === 0 ? 'Even' : 'Odd';
// 'Even' is assigned to the msg variable
// since the condition evaluates true.
```

Operator Precedence

Some operators are applied before others, when calculating the result, this is called operator precedence. The operator precedence in JavaScript is the same as was taught in math.

*The calculation is always evaluated from **left to right**.*

Operator precedence determines how operators are parsed concerning each other. Operators with higher precedence become the operands of operators with lower precedence.

```
/*
Evaluate: 50 + 10 / 8 + 2
Step 1: 50 + 10 / 8 + 2
Step 2: 50 + 1.25 + 2
Step 3:  51.25 + 2
Result: 53.25
*/
```

Operator Associativity

When multiple operators in expression have the same precedence, then JavaScript groups them by associativity. JavaScript have 2 types of associativity:

1. **Left-Associativity:** *left-to-right* associativity where expression is interpreted as:

   (*operandA *Operator1* operandB*) *Operator2* operandC

2. **Right-Associativity:** right-to-left associativity where expression is interpreted as:

   operandA *Operator1* ***(*operandB *Operator2* operandC*)***

*The **assignment** and **exponential** operators have **right-associativity**.*

```
1  const a = b = 5; // Same as:  a = (b = 5);
2  const a = 4 ** 3 ** 2; // Same as: 4 ** (3 ** 2);
```

*Other **arithmetic** operators have **left-associativity**.*

```
1  const a = 4 / 3 / 2; // Same as: (4/3) / 2;
```

Summary

- JavaScript has **Unary**, **Binary** and **Ternary** operators.
- All **Binary** operators are **infix** expressions, where the **operator** is placed between **two operands**.
- Binary operators can be categorized under: *Arithmetic*, *Relational*, *Equality*, *Assignment*, *Logical* and *Bitwise* operators.
- **Type Coercion** is automatic/implicit conversion of value from one type to another.
- **Strict Equality/InEquality** operators check operands without *type-coercion*.
- Logical AND (&&) operator with **non-boolean** values, returns ***operand1*** if it can be converted to ***false***, otherwise returns ***operand2***.
- Logical OR (||) operator with **non-boolean** values, returns ***operand1***, if it can be converted to ***true***, otherwise returns ***operand2***.
- **Short-circuit** can be used for conditional evaluation.
- The ***delete*** operator deletes object property, but can't delete the entire object or *non-configurable* object.
- The ***typeof*** operator returns a string indicating the **type** of operand.
- The **ternary operator** takes three operands.
- JavaScript also provides ***void***, ***in***, ***instanceOf*** operators.
- Calculations are always evaluated from **left to right** under defined **operator precedence**.

CHAPTER 6

Control Structures

Preface to Control Structures

Introduction

Conditional and Iteration Statements

Conditional Statements

if-else, switch-case, Ternary Operator (?:)

Iteration Statements

while, do-while, for, for...of, for...in loops

Control Structures

JavaScript provides many control structures to control the flow of execution. Control structures can be divided in two types of statements:

- **Condition Statements**
  if-else, switch-case, ternary operator (?:)

- **Looping/Iteration Statements**
  while, do-while, for, for...of, for...in

Conditional Statements

if-else

This is used to check conditions and execute either *true* block or *false* block statements.

```
let num = 5;
if (num % 2 === 0) {
  console.log('Number is Even');
} else {
  console.log('Number is Odd');
}
```

If-else can be nested.

```
let num = 5;
if (num % 2 === 0) {
  console.log('Number is Even');
} else if (num < 2) {
  console.log('Odd, less than 2');
} else {
  console.log('Odd, greater than 2');
}
```

switch-case

This can be used to check the *number* or *string* value against a given set of values. In simple words, it matches a given value against a set of values.

```
let day = 2;
let dayName = '';
switch (day) {
  case 1:
    dayName = 'Monday';
    break;
  case 2:
    dayName = 'Tuesday';
    break;
  // …
  // …
  case 7:
    dayName = 'Sunday';
    break;
  default:
    dayName = 'Invalid Day';
}
```

case statements are matched with the given value (day). Switch stops matching on the first match.

break statement is used to *terminate* the switch after first selection. If we *remove break* statements from between the cases, in that scenario, all cases after the first match will be executed.

This is because the switch stops checking/matching after the first match, so all following cases will be considered a match.

default statement is used to define the default case, which will be executed, if none of the cases match with the given value. The *default* case is optional.

Ternary Operator (?:)

Used for checking simple expressions.

```
let num = 10;
const msg = num % 2 === 0 ? 'Even' : 'Odd';
// 'Even' is assigned to the msg variable
// since the condition evaluates true.
```

Looping/Iteration statements

Iteration statements are used to iterate or loop a given set of statements until termination statement for the loop is reached. All loops consists of three expressions:

1. **Initialization expression**: Defines the starting point of the loop.
2. **Termination expression**: Defines the end point of the loop.
3. **Update expression**: Defined step value for the loop.

while loop

while loop does not have surety, whether it will execute at least once or not. Iteration of the statement could be skipped, if the termination expression is met on the first iteration.

```
let num = 1; // Initialization expression
while (num <= 5) {
  // Termination expression
  console.log(num);
  num++; // Update expression
}
//The above code will print numbers from 1 to 5.
```

do-while loop

Similar to *while* loop, with one difference that it gives the surety that the body of loop will execute **at least once**.

```
let num = 1; // Initialization expression
do {
  console.log(num);
  num++; // Update expression
} while (num <= 5); // Termination expression
//The above code will print numbers from 1 to 5.
```

for loop

Similar to *while* loop, except that initialization, termination and update expressions are written in a single line.

```
for (let num = 1; num <= 5; i++) {
  console.log(num);
}
//The above code will print numbers from 1 to 5.
```

for...of loop

This creates a loop iterating over ***iterable objects***, including built-in ***String***, ***Array***, **array like objects** (like, *arguments* or *NodeList*), ***TypedArray***, ***Map***, ***Set*** and *user defined iterables*.

```
const arr = ['a', 'b', 'c'];
for (const itm of arr) {
  console.log(itm);
}

//The above code prints on console:
// 'a'
// 'b'
// 'c'
```

for...in loop

Used for iterating over all enumerable properties of an object with valid keys (ignoring properties with *Symbol()* keys), including inherited enumerable properties.

```
const obj = { a: 10, b: 20, c: 30 };
for (const itm in obj) {
  console.log(`${itm} : ${obj[itm]}`);
}

// The above code prints on console:
// 'a : 10'
// 'b : 20'
// 'c : 30'
```

Note: loops can be terminated forcefully using ***break***, ***throw*** or ***return***.

Summary

- JavaScript provides **Conditional** and **Looping** controls structures.
- The ***if-else*** checks the condition and executes either *if* or *else* block.
- The ***switch-case*** checks the *number/string* value against a given set of values.
- The ***while***, ***do-while***, ***for***, ***for...of***, ***for..in*** loops can be used for looping/iterating the iterable values.
- Loops can be terminated **forcefully** using ***break***, ***throw*** and ***return*** statements.

CHAPTER 7

Objects

Preface to Objects

Introduction

Group of key-value pairs, constructor method, object literal

Accessing Object Properties

Dot Notation, Bracket Notation

Objects

Objects are a way of grouping similar/meaningful data together. JavaScript objects are a **group of *key-value* pairs**.

The **"key"** part is a *string* value, while **"value"** could be any valid JavaScript value. There are two ways of creating an object:

1. Using Object constructor method

new Object({<initial values>})

```
const obj = new Object({
  name: 'JavaScript', version: 6
});
```

2. Using object literal syntax

```
const obj = { name: 'JavaScript', version: 6 };
```

Accessing object properties

There are two ways of accessing properties of an object:

1. Using Dot Notation

This is considered best practice to access properties by **"dot notation"**, because this can be optimized by the JavaScript engine as well as the minifiers for performance benefits.

Syntax

<objectName>.<property-key>

```
1  const obj = { name: 'JavaScript', version: 6 };
2  const version = obj.version; // 6
```

2. Using Bracket Notation

The advantage of **"bracket notation"** is that ***key*** values can be calculated at run time, since *keys* are provided as *strings*. Another advantage is that ***reserved keywords*** can be used as property *keys*.

Although, this has a negative impact on performance, because this prevents optimizations that can be applied by JavaScript engine and minifiers.

Syntax

<objectName>[<property-key>]

```
1  const obj = { name: 'JavaScript', version: 6 };
2  const version = obj['version']; // 6
```

Note: Starting in ECMAScript 5, reserved keywords may be used as property/attribute key, without wrapping them in quotes (“”).

const obj = { for : 10 };

“*for*” is a reserved keyword, but now can be used as a property key.

Note: Starting in ECMASCript 2015, object keys can be defined by the variable using bracket notation on being created.

let versionKey = "version_key"

const obj = { [versionKey] : 6 };

Summary

- **Objects** are a group of **key-value pairs** for grouping similar data together.
- The **"key"** should be a *string*, while **"value"** can be any valid value.
- Objects can be created using **constructor** or **literal syntax**.
- The **dot notation** is preferred for accessing properties, because these can be optimized for performance benefits.
- The **"key"** can be calculated at **runtime**, if using **bracket notation**, although it has a negative impact on performance.

CHAPTER 8

Arrays

Preface to Arrays

Introduction

Group of similar/meaningful data, constructor method, array literal

Accessing Array Items

Zero based index, using loops to iterate array items

Array Methods

toString(), concat(), join(), pop(), push(), unshift(), shift(), slice(), splice(), sort(), reverse()

Arrays

Arrays in JavaScript are actually a special type of object. Similar to Object, Arrays are also a way to group similar/meaningful data together. There are two ways of creating an array:

1. Using Array constructor method

Syntax
new Array(<list-of-initial-values>)

```
const arr = new Array('JavaScript', 6);
```

2. Using array literal syntax

```
const arr = ['JavaScript', 6];
```

Array items are stored on consecutive locations with a zero based index, i.e, the first item of the array is stored at index 0 (zero) and last element at length -1 index.

"JavaScript" **index: 0**	6 **index: 1**

Accessing Array Items

The ***length*** property of the array holds the *highest/last index of the array + 1*. This isn't necessarily the number of items in the array.

```
const arrItems = ['cat', 'dog'];
arrItems[100] = 'fox';

arrItems.length; // 101
// Highest index in the array + 1
```

There are multiple ways of accessing/iterating *array* items. For example, *for*, *for...of*, *while*, *do-while*, *Array.forEach*, *Array.map*, etc.

```
// Using for loop
for (let i = 0; i < arr.length; i++) {
  console.log(arr[i]);
}

// Using Array.map()
arr.map((itm) => {
  console.log(itm);
});
```

Array Methods

Let's learn about a few of the methods of the Array.

toString()

Returns a *string* with the *toString()* of each element separated by commas.

```
let arr = ['cat', 'dog'];
arr.toString();
// 'cat, dog'
```

concat()

Return a new *array* with the given items added on to it.

```
let arr = ['cat', 'dog'];
arr.concat('fox', 'tiger');
// ['cat', 'dog', 'fox, 'tiger']
```

join(separator)

Converts the *array* to a *string*, with values delimited by the given **'separator'**.

```
let arr = ['cat', 'dog', 'fox'];
arr.join('#');
// 'cat#dog#fox'
```

pop()

Removes and returns the last item

```
let arr = ['cat', 'dog', 'fox'];
const lastItem = arr.pop();
console.log(lastItem); // 'fox'
console.log(arr); // ['cat', 'dog']
```

push()

Appends items to the end of the *arr*ay.

```
let arr = ['cat', 'dog'];
arr.push('lion', 'elephant');
// ['cat', 'dog', 'lion', 'elephant']
```

unshift()

Prepends items to the start of the *array*.

```
let arr = ['cat', 'dog'];
arr.unshift('rat');
// ['rat', 'cat', 'dog']
```

shift()

Removes and returns the first item of the array.

```
let arr = ['cat', 'dog','lion'];
const firstElement = arr.shift();
console.log(firstElement);  // 'cat'
console.log(arr); // [ 'dog', 'lion']
```

slice()

Returns a sub-array of the *array*. It returns a *shallow copy* of the portion of an *array* into a new *array*, selected from **"start"** index to **"end"** index (*"end"* index not included). The array will **not be modified**.

Syntax

slice();

*slice(**start**);*

*slice(**start**, **end**);*

start: Zero based index of the starting items for the extraction.

```
const arr = ['cat', 'dog', 'lion', 'fox', 'tiger'];
arr.slice(2); // ['lion', 'fox', 'tiger']
// Extracting all elements from index 2 to end of the array.
```

A **negative start index** can be used, indicating an offset from the end of the *array*.

```
const arr = ['cat', 'dog', 'lion', 'fox', 'tiger'];
arr.slice(-1); // ['elephant']
//Extracting one element from the end of the array,
// i.e., extracting the last item of the array.
```

If **start** is undefined, slice starts from index 0.

```
const arr = ['cat', 'dog', 'lion', 'fox', 'tiger'];
arr.slice(); // ['cat', 'dog', 'fox', 'lion', 'tiger']
```

If **start** is greater than the index range of the *array*, an empty *array* is returned.

```
const arr = ['cat', 'dog', 'lion', 'fox', 'tiger'];
arr.slice(10);  // []
```

end: Index of the first element to exclude from the returned array. *slice()* extracts up to but not including the **"end"** index.

```
const arr = ['cat', 'dog', 'lion', 'fox', 'tiger'];
arr.slice(1, 4); // ['dog', 'lion', 'fox']
// Extracts elements from index 1 up to
// index 3 (end index - 1).
```

A **negative end index** can be used, indicating an offset from the end of the *array*.

```
const arr = ['cat', 'dog', 'lion', 'fox', 'tiger'];
arr.slice(1, -2); // ['dog', 'lion']
// Extracting from start index 1 to excluding
// the last 2 elements from the end.
```

If **end** is omitted, *slice()* extracts through end of the array, i.e., *arr.length*.

```
const arr = ['cat', 'dog', 'lion', 'fox', 'tiger'];
arr.slice(2); // [ 'lion','fox', 'tiger']
// Extracting from start index 2 to end of the array.
```

If the **end** is greater than the length of the array, *slice()* extracts through the end of the array i.e., end becomes the length of the array.

```
const arr = ['cat', 'dog', 'lion', 'fox', 'tiger'];
arr.slice(2, 10); // [ 'lion', 'fox', 'tiger']
// Extracting from start index 2 to end of the array,
// because given end was greater than the array length,
// therefore end becomes 5 (length of the array).
```

splice()

Modifies the existing Array by deleting a section and replacing it with more items.

Syntax

splice(start, numberOfItemsToRemove, itemToAdd);

start: Zero based index of the starting items for the deletion/replacement.

numberOfItemsToRemove: Number of elements to be removed.

itemToAdd: *optional* item to be added after the start index.

```
const arr = ['cat', 'dog', 'lion', 'fox', 'tiger'];
arr.splice(2, 0, 'rabbit');
// ['cat','dog', 'rabbit', 'lion', 'fox', 'tiger']
// Starting from index 2, remove 0 (zero) items
// and add “rabbit” after start index 2.
```

```
const arr = ['cat', 'dog', 'lion', 'fox', 'tiger'];
arr.splice(2, 2, 'turtle');

console.log(arr);
// ['cat', 'dog', 'turtle', 'tiger']
// Starting from index 2, remove 2 items and
// add 'turtle' after start index 2.
```

sort()

Sorts the given *array*. This takes an *optional* comparison function.

```
const arr = ['cat', 'dog', 'lion', 'fox', 'tiger'];
arr.sort(); // ['cat', 'dog', 'fox', 'lion', 'tiger']
```

reverse()

Reverses the order of items of the *array*.

```
const arr = ['cat', 'dog', 'lion', 'fox', 'tiger'];
arr.reverse(); // ['tiger', 'fox', 'lion', 'dog', 'cat']
```

Summary

- Array items are stored in consecutive *locations starting* with **zero index**.
- Arrays can be created using **constructor** or **literal syntax**.
- The *while*, *do-while*, *for, map* and other loops can be used for accessing array items.
- Arrays provide useful methods like: *toString(), push(), pop(), concat(), join(), shift(), unshift(), slice(), splice(), sort(), reverse()*, etc.

CHAPTER 9

Functions

Preface to Functions

Introduction

Function Declaration, Function Expression, Function Invocation, arguments, Rest Parameters

Anonymous Functions

Lambda Functions without name, transient functions

IIFE

Declaring and Invoking function in a single expression, Self-Executing Functions

Inner Functions

Nested functions, access parent scopes in child functions

Functions

Functions are one of the core components of JavaScript. They are used to define a group of statements, so that they can be reused multiple times.

They help reduce code size by encapsulating reusable code. Functions make code more readable/manageable by breaking code in logical blocks.

There are two ways of creating a function:

- **Function declaration**
- **Function expression**

Function declaration

function keyword is used to declare a function.

Syntax

```
function <name-of-function>(<parameters>) {
        ...
        <body of the function>
}
```

name-of-function: A valid identifier/name for the function.

parameters: These are named parameters/arguments. A function can have 0 (zero) or more parameters.

```
1  function add(a, b) {
2    return a + b;
3  }
```

<name-of-function> is **"add"**, while *<parameters>* are "**a**" and "**b**".

return: statement is used to return any valid value from the *function*. This value is returned to the *caller* of the function, terminating the function. If no *return* statement is given, then JavaScript returns *undefined*.

Function expression

This is very similar to function declaration and has almost the same syntax.

The main difference is the **name-of-function**, which can be omitted to create ***anonymous functions***. Another difference is that ***function expressions are not hoisted***, while function declarations are ***hoisted***.

Syntax

*const <**name-of-function**> = **function**(<**parameters**>) {*
...
<body of the function>
}

```
1  const add = function (a, b) {
2    return a + b;
3  }
```

Function Invocation/Calling

Syntax
functionName(parameters);

functionName: Name of the function to be called.
parameters: Parameters are assigned from left to right.

```
1  const result = add(2, 5);
2  console.log(result); // 7
```

*The function **"add()"** is called with parameters **2** and **5**. Since parameters are assigned from left to right, therefore, 2 is assigned to parameters "**a**" and 5 assigned to **"b"**. Function **returns** sum of both the values and result is assigned a value of **7**.*

Note: We can pass more **parameters** than the function is expecting, all extra parameters are ignored.
add(2, 5, 10); // **10 will be ignored**

arguments

The **"arguments"** is an Array-like *object* accessible inside a *function*, that contains the values of the parameters passed to the functions. This can be considered as a *special object* containing all *parameters* passed to the function at the time of *calling/invocation*.

```
1  function add() {
2    let sum = 0;
3    for (const itm of arguments) {
4      sum += itm;
5    }
6    return sum;
7  }
8
9  const result = add(1, 2, 5, 10); // 18
```

*"**add**" function can now accept any number of arguments, the keyword **arguments** will automatically contain all the parameters passed at the time of function call/invocation.*

> **Note**: JavaScript functions are **variadic**, i.e., they can take any number of arguments. Functions are treated as **first-class-citizens**, because they can be used as other values. For example, they can be assigned to a variable, they can be passed as parameters to other functions, basically we can do all actions, which are applicable to other types.

Rest parameters

Using **arguments** *object* is pretty useful, but it does seem a little *verbose*, because by looking at the function *declaration/expression*, we can't predict whether the function expects any *argument* or not. To make functions more readable, we can replace ***arguments*** with **"rest parameters"**.

Rest parameters allows a function to accept an indefinite number of arguments. This was introduced in **ES6**.

Same add() function with the use of rest parameters.

```
function add(...args) {
  let sum = 0;
  for (const itm of args) {
    sum += itm;
  }
  return sum;
}

const result = add(1, 2, 5, 10); // 18
```

Note: It is important to note that wherever the rest parameters are placed in function declaration, it will store all arguments **after** its declaration, but **not before**.

For example:
function add(firstValue, ...args) { }
add(1, 2, 5, 10);
1 will be assigned to the "**firstValue"** parameter, while **2**, **5** and **10** will be assigned to the rest parameter **"args"**.

Anonymous Functions

Anonymous functions are one, which do not have any name. These are also known as **"lambda functions"**. These are usually transient functions that are either passed into or returned from another function.

It allows us to write a function inline with its usage, instead of declaring it somewhere else in the code.

```
let arr = ['cat', 'dog', 'rabbit'];
arr.map((itm) => {
  console.log(itm);
});

// Output:
// 'cat'
// 'dog'
// 'rabbit'
```

IIFE (Immediately Invoked Function Expression)

JavaScript provides a mechanism for simultaneously declaring and invoking a function, using a single expression, it is called **IIFE**.

Syntax

```
(function() {
        ...
        // body of the function
}) ( );
```

It is a *design pattern* which is also known as a **"Self-Executing Anonymous Function"** and contains two major parts:

1. First is the anonymous function with ***lexical scope***. This prevents accessing variables within the IIFE scope from outside code, as well as avoids polluting global scope.

2. The second part "**( )"** (*Parenthesis, placed at the end of the function declaration*)**,** creates the immediately invoked function expression, through which the JavaScript engine will directly interpret the function.

Inner Functions

JavaScript function *declarations/expressions* are allowed inside another function. An important concept of nested functions is that they can ***access variables of the parent function's scope***.

Inner/nested functions help writing more maintainable code. If a *function* relies on one or more *functions* that are

not used to any other part of the code, such *functions* can be nested as utility functions inside the ***main function***.

This keeps the number of global functions down, which is always considered as a best practice.

```
function parentFunc() {
  const a = 1;
  function innerFunc() {
    const b = 4;
    return a + b;
    // variable 'a' can be accessed here
  }

  return innerFunc();
}
```

Summary

- Functions make code more manageable by splitting code in logical blocks.
- Functions can be created using: **Function Declaration** and **Function Expression**.
- Function can have **zero** or **more arguments/parameters**.
- Extra parameters given to function calls are ignored.
- The ***"argument"*** is a keyword which holds *all the parameters* of a function in an *Array-like object*. It seems verbose, since it hides the actual declaration of parameters.
- **Rest parameters** allow the function to accept an indefinite number of arguments.
- **Anonymous functions** also called **"lambda functions"** do not have names.
- **IIFE** is used to declare and invoke a function simultaneously.
- **Inner functions** are functions inside functions, which helps reduce unnecessary global functions.

CHAPTER 10

Closures

Preface to Closures

Introduction

Closure is combination of function and surrounding state

Lexical Scope

Uses location to define scope, Global/Function/Lexical Scope, Private members

Scope Chain

Closure has Local, Enclosing and Global Scope, live-binding, problem creating closures in loop

Closures

A **closure** is the ***combination*** of a ***function bundled together*** (enclosed) with the references to its **surrounding state** (the **lexical** *environment variables*).

In other words, a closure gives us access to an outer function's scope from an inner function, ***when the parent function has returned***. In JavaScript closures are ***created every time a function is created***.

*Let's learn about **Lexical Scope**, in order to understand closures better.*

Lexical Scope

The word **"lexical"** refers to the fact that lexical scoping uses the **location** where a variable is declared within the source code, to determine where that variable can be ***accessible***.

There are three types of scope in JavaScript:

1. **Global Scope**
   Variables defined in global scope are available throughout the source code.

2. **Function Scope**
   Variables defined in function scope are available in that function body only i.e they are local to that function.

3. **Lexical Scope**
   This is also known as block-scope and was introduced in ES6. Variables defined in lexical scope are available only within the enclosing block.

In ES6, JavaScript introduced the **let** and **const** keywords, which allows to create block-scoped variables.

```
1  if (true) {  const x = 1; }
2  else { const x = 2; }
3  console.log(x); // ReferenceError
```

The above statement gives ReferenceError, because "x" is a block-scoped variable and is not accessible here.

In contrast to the above example, if we declare a variable using ***var***, it would create a global scope and the variable should be available after the *if-else* block.

```
1  if(true) { var x = 1; }
2  else { var x = 2; }
3  console.log(x); // 1
```

Since we know about lexical scope, now let's try to understand closures.

In closure, the *lexical environment* consists of any local variables that were *in-scope* at the time of closure creation. *Let's try to understand this with the help of an example.*

```
1  function makeAdder(x) {
2    return function (y) {
3      return x + y;
4    };
5  }
```

Lexical environment contains: variable **x**

```
1  const add5 = makeAdder(5);
```

*Because the **makerAdder()** function is returned, **"x"** should have been gone, but since it formed closure, "x" is still available in **lexical scope**.* The value of the lexical environment variable **x** would be **5**.

```
1  const add10 = makeAdder(10);
```

*Similarly, in the above code, the value of the lexical environment variable **x** would be **10**.*

```
1  console.log(add5(2)); // 7
2  console.log(add10(2)); // 12
```

Both ***add5()*** and ***add10()*** are closures. They share the same function body definition, but store different lexical environments. In ***add5()***, lexical environment **x** is 5, while in ***add10()*** it is 10.

In essence, ***makeAdder()*** is a **function factory**. It creates functions that can add a specific value to their arguments.

One of the main use cases for closures is creating *private methods/properties* in functions.

Creating private methods/properties using Closures

JavaScript, prior to ***classes***, didn't have a native way of declaring private methods, but it was possible to emulate private methods using closures.

```
const makeCounter = function () {
  // private members
  let privateCtr = 0;
  function changeBy(val) {
    privateCtr += val;
  }

  // public members, returned from here
  return {
    increment() {
      changeBy(1);
    },
    decrement() {
      changeBy(-1);
    },
    value() {
      return privateCtr;
    }
  };
};

const counter = makeCounter();
counter.value(); // 0
counter.increment();
counter.value(); // 1
counter.decrement();
counter.value(); // 1
```

The lexical environment contains two private members:

1. **privateCtr** variable
2. **changeBy()** function

The private members can't be accessed from outside the *makeCounter()* function. Instead, public members should be used to access the private members.

Below are the public functions returned from *makeCounter()*:

1. **increment()** function
2. **decrement()** function
3. **value()** function

Above public functions are closures that share the same lexical environment and because of lexical scoping, they each have access to the ***privateCtr*** variable and the ***changeBy*** function.

Closure Scope Chain

Every closure has three scopes:

1. **Local Scope** (Own scope)
2. **Enclosing Scope** (Can be block, function or module scope)
3. **Global Scope**

When nested functions themselves contain nested functions, this effectively creates a chain of function scopes.

```
const globalVal = 10; // Global Scope
function sum(a) {
  // outer scope
  return function (b) {
    // outer scope
    return function (c) {
      // local scope
      return a + b + c + globalVal;
    };
  };
}

const result = sum(1)(2)(3);
console.log(result); // 16
```

There's a series of nested functions, all of which have access to the outer scope. In this context, it can be said that closures have access to all outer function scopes.

Closures can access members in ***module scope*** too. We'll learn about **modules** in **Chapter 10**, but for now, let's just try to understand module scope with closures.

```
// my-module.js
let x = 5;
export const getX = () => x;
export const setX = (val) => {  x = val; };
```

Here, the module *exports* a pair of *getter-setter* functions, which close over the module-scoped variable **"x"**. Even when **"x"** is not directly accessible from other modules, it can be read/written with the help of exported functions.

```
import { getX, setX } from './my-module.js';
console.log(getX()); // 5

setX(6);
console.log(getX()); // 6
```

Closures can close over imported values as well, which are regarded as **"live bindings"**, because when the original value changes, the imported members changes automatically.

```
// my-module.js
export let x = 1;
export const setX = (val) => {  x = val; };

// closure-creator.js
import { x } from './my-module.js';

export const getX = () => x;
// Live binding is created for variable 'x'

// main-program.js
import { getX } from './closure-creator.js';
import { setX } from './my-module.js';
console.log(getX()); // 1

setX(2);
console.log(getX()); // 2
```

The setX(2) function call (line 16) *updated the value of* ***"x"****, placed in my-module.js, and at the same time updated the value of* ***"x"*** *in closure-creator.js with the help of* ***live binding****.*

Problem creating Closures in loop

Prior to the introduction of the ***let*** keyword, a common problem with closures occurred when closures were created inside a loop. Let's try to explore the problem and then understand the possible solutions.

```
// HTML required for the example
<p id='help'>Helpful notes will appear here<p>
<p>Email: <input type='text' id='email' name='email' /> <p>
<p>Name: <input type='text' id='name' name='age' /> <p>
<p>Age:     <input type='text' id='age' name='age' /> <p>
```

```
function getElement(id) {
  return document.getElementById(id);
}
function showHelp(help) {
  getElement('help').textContent = help;
}

var helpText = [
  { id: 'email', help: 'Your email address' },
  { id: 'name', help: 'Your name' },
  { id: 'age', help: 'Your age' }
];

function setupHelp() {
  for (var i = 0; i < helpText.length; i++) {
    var item = helpText[i];
    getElement(item.id).onfocus = function () {
      showHelp(item.help);
    };
  }
}

setupHelp();
```

When the above code runs, we'll see that no matter what field we *focus* on, the help text of your **"age"** is displayed.

The reason for this is that the functions assigned to ***onfocus*** are ***closures***, they consist of the function definition and the captured lexical environment from the *setupHelp()* function's scope.

Three closures have been created by the loop, but each of them share the same lexical environment, which has a variable with changing values (*"item"*). Because the variable **"item"** is declared with ***var***, it has *function scope*. The value of **"item.help"** is determined when the *onfocus* callbacks are executed.

But, since the loop has already run its course by that time, the **"item"** variable (shared by all closures) has been left pointing to the last entry in the ***helpText*** array. Therefore, it will always display the ***last*** help text.

Solutions to the above problem

Let's see possible solutions with and without ES6's **let** keyword.

Solution 1:

By creating more closures, in particular, by using a ***function factory***.

```
function makeHelpCallback(help) {
  return function () {
    showHelp(help);
  };
}
```

*Rather than the callbacks all sharing a single lexical environment, the **makeHelpCallback()** function creates a new lexical environment for each callback, in which **"help"** refers to the corresponding item from the **helpText** array.*

```
// Rewriting setupHelp to include the solution
function setupHelp() {
  for (var i = 0; i < helpText.length; i++) {
    var item = helpText[i];
    getElement(item.id).onfocus = makeHelpCallback(item.help);
  }
}
```

Solution 2:

By using *anonymous closures* with the help of **IIFE** (Immediately Invoked Function Expression).

```
// Rewriting setupHelp to include the solution
function setupHelp() {
  for (var i = 0; i < helpText.length; i++) {
    (function () {
      var item = helpText[i];
      getElement(item.id).onfocus = function () {
        showHelp(item.help);
      };
    })();
  }
}
```

Solution 3:

By using ES6's ***let*** or ***const*** keyword.

The use of *let* or *const* will create block scope, hence fixing the problem.

```
// Rewriting setupHelp to include the solution
function setupHelp() {
  for (let i = 0; i < helpText.length; i++) {
    let item = helpText[i];
    getElement(item.id).onfocus = () => {
      showHelp(item.help);
    };
  }
}
```

Summary

- **Closure** is the combination of *function* and the *surrounding state*.
- The **Lexical scope** uses the **location** of the variable to define its *scope*.
- Closures can be used to declare **private members** of the function.
- Every closure has *Local*, *Enclosing* and *Global* scope.
- **Function factory**, **IIFE** and ***let/const*** can be used to fix closure issues when used in a loop.

CHAPTER 11

Arrow Functions

Preface to Arrow Functions

Introduction

Function that retains lexical context, No binding of 'arguments' Limitations

Arrow Functions with class

Declare class fields, bind(), call(), apply()

Arrow Function Evaluation

Returning Object Literals, Handling Line Breaks, Parsing Order

Arrow functions

In *arrow functions*, this retains the value of the enclosing lexical context's ***this***. In global code, ***this*** will be set to **global** object.

> **Note**: If **thisArg** is passed to ***call()***, ***bind()*** or ***apply()*** on invocation of an arrow function, it will be **ignored**. We can still **prepend arguments** to the method, but the first argument (**thisArg**) should be set to ***null***.

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

Syntax

```
(arguments) => {
        // statements
}
```

arguments: Similar to traditional function, it is a list of input parameters passed to the function.

=> : Also called Fat Arrow operator.

```
1  const sum = (num1, num2) => {
2    return num1 + num2;
3  };
4
5  sum(2, 5); // 7
```

this refers to the lexical ***context's this***.

```
var a = 100;
const obj = {
  a: 10,
  show: () => { console.log(this.a); }
};

obj.show(); // 100
```

*Above code prints 100 because lexical context is the **global object**, which has a variable named **"a"** with value **100**.*

If we use a traditional function, ***this*** will refer to the given object **"obj"**.

```
var a = 100;
const obj = {
  a: 10,
  show: function () { console.log(this.a); }
};
obj.show(); // 10
// Because this refers to the given object 'obj'.
```

Limitations of Arrow function

There are some limitations with *arrow function* as compared to *traditional function*:

1. Arrow functions don't have their own bindings to ***this***, ***arguments*** or ***super***.
2. Arrow functions don't have access to the ***new.target*** keyword.

3. Arrow functions aren't suitable for ***call***, ***apply*** and ***bind*** methods, which generally rely on establishing a scope.
4. Arrow functions can't be used as ***constructors***, it will throw an error when used with the ***new*** keyword.
5. Arrow functions can't use ***yield***, within its body.
6. Arrow functions do not have a ***prototype*** property.

Let's learn how we can decompose traditional function to arrow function.

```
// Traditional function
function isEven(num) { return num % 2 === 0; }
```

Below compact alternative is known as an *arrow* **function with a block body**.

```
const isEven = (num) => { return num % 2 === 0 };
```

Even more decomposed arrow function, this compact alternative is known as the ***arrow function with a concise body***.

```
const isEven = (num) => num % 2 === 0;
```

Below changes were made to make it more concise:

- The *parentheses* are **removed** around the argument ***"num"***, parentheses can be removed if the function has ***only one argument***.
- The *return* statement is **removed**, this can be done if the function contains **only one statement**.

Parentheses around arguments cannot be removed for below cases:

- If the arrow function has more than one argument.
- ***Rest parameters*** always require parentheses.
  *(a, b, **...obj**) => expression*
- ***Default parameters*** always require parentheses.
  *(**a = 10**, **b = 5**, c) => expression*

Arrow functions as class fields

A class's body has ***this*** context, *arrow functions* as ***class fields*** uses the class's ***this*** context and ***this*** inside arrow function will point to the *instance (or the class itself, for static fields)*.

> **Note**: Static fields cannot be directly accessed on the instances of the *class*. Instead, accessed on the *class* itself.

However, because it created a *closure*, not the function's *own binding*, the value of ***this*** will not change based on the execution context.

Because the class's arrow function always refers to **this** of the class, which **never changes**, arrow functions are often said to be **"auto-bound methods"**.

```
class User {
  name = 'JavaScript';
  showName = () => {
    console.log(this.name);
  };
}

const user = new User();
user.showName(); // 'JavaScript'

const { showName } = user;
showName(); // 'JavaScript'
```

*Because the class's arrow function always refers to **this** of the class, which **never changes**. Therefore **"JavaScript"** was printed on the console, otherwise undefined should have been printed.*

Arrow function not to be used with call, apply and bind

The ***call()***, ***apply()*** and ***bind()*** methods are not suitable as arrow functions, as these methods were designed to allow methods to execute with ***different scopes***, but arrow functions on the other hand establish ***this*** only based on the scope of the context in which the arrow function is defined.

```
const obj = { a: 10 };
var a = 100;
const sum = (num1, num2) => this.a + num1 + num2;

sum.call(obj, 5, 50); // 155
//Because: 100    +  5   +   50 = 155
//        this.a    num1    num2
```

*In the last code, although the expected result was **65** (10 + 5 + 50), we got **155**, proving that arrow functions point **this** to the enclosing scope, which is the **window** here.*

Same behavior is observed with ***bind()*** and ***apply()*** methods.

> **Note**: Class fields are defined on the **instance**, not on the **prototype**, so every instance creation would create a new function reference and allocate a new closure, potentially leading to more memory usage than a normal unbound method.

No binding of "arguments" object

The arrow functions do not have their own ***"arguments"*** object.

```
1  const show = () => arguments[0];
2  show(); // 'ReferenceError: arguments is not defined'
```

***arguments*:** This is a reserved keyword which holds the list of arguments passed to the function.

Returning "Object Literals"

Returning ***object literals*** from arrow functions with a ***concise body*** will not work as expected.

```
1  const user = () => { name: 'JavaScript'; };
2  user(); // undefined
```

*Because JavaScript only sess the arrow function as having a concise body if the token following the arrow is not a left parentheses, so the code inside curly braces ({}) is **parsed as a sequence of statements** (i.e., "name" is treated like a **label** and not as a **key** in object literal).*

Correct way to return object literal would be:

```
1  const user = () => ({ name: 'JavaScript' });
```

Line breaks in arrow functions

An arrow function cannot contain a line break between its parameters and its arrow.

```
1  const user = (name)
2      => 'JavaScript';
3  //This will give SyntaxError: Unexpected token '=>'
```

However, line breaks can be added after the arrow or using parentheses.

```
1  // Valid
2  const user = (name) =>
3          'JavaScript';
```

Parsing Order

Although the ***fat arrow*** (=>) in an arrow function is not an ***operator***, arrow functions have special parsing rules that

interact differently with ***operator precedence*** compared to traditional functions.

```
let defaultFunc;
defaultFunc = defaultFunc || () => { };
// SyntaxError: invalid arrow function arguments
```

The SyntaxError is because => has a lower precedence than other operators.

Correct way would be:
Parentheses are necessary to avoid **defaultFunc || ()** being parsed as the arguments of the arrow function.

```
let defaultFunc;
defaultFunc = defaultFunc || (() => {});
```

Summary

- In arrow function, **"this"** refers to the lexical context's **"this"**.
- Arrow functions don't have their own bindings to ***this***, ***arguments*** or ***super***.
- Arrow functions are not suitable for ***call***, ***apply*** and ***bind*** methods.
- Arrow functions can't be used as ***constructor*** and they do not have a ***prototype***.
- Arrow functions can be declared with **block** or **concise** body.
- Parentheses around *arguments* are needed for: more than one argument, *rest parameters* and *default parameters*.
- Arrow function can't have **line breaks** between *parameters* and *arrow*.

CHAPTER 12

Modules

Preface to Modules

Introduction

Splitting JavaScript programs into separate modules

export Statement

Named & Default Export, Renaming, Re-exporting, Aggregation, Wild-Card Export

import Statement

Named & Default Import, Namespace, Side-Effect import

Modules

JavaScript programs started off with small usage, primarily to add interactivity to web pages. But, now we have complete web applications written in JavaScript.

An application would need a gigantic size of code to be written and then loading that gigantic code will impact the performance, impacting the user experience. Therefore some mechanism was required for splitting JavaScript programs into separate modules that can be imported when needed.

Node.js has had this ability for a long time and there are a number of JavaScript libraries and frameworks that enable module usage. For example: *CommonJs*, *RequireJs*, *Webpack*, etc.

Modern browsers have started supporting module functionality natively, which means, browsers can optimize loading of modules, making it more efficient than having to use a library/framework to do all the required stuff. Use of native JavaScript modules is dependent on the ***import*** and ***export*** statement.

export statement

The **export** declaration is used to export values from a JavaScript module. Exported modules can then be imported into other programs with the **import declaration** or **dynamic import**.

When a module updates the value of a binding that is exported, the update will be visible in its imported value. This update mechanism is called **live binding**.

In order to use the *export* declaration, the file must be interpreted by the runtime as a **module**. Modules are automatically interpreted in **strict mode**.

In HTML, this is done by adding the ***type="module"*** attribute to the ***<script>*** tag.

Every module can have two types of export:

- **Named export**
- **Default export**

Named Export

As the name suggests, named exports are ones which have a unique name. A module can have multiple named exports.

While importing, named exports must be referred to by the exact same name (*optionally renaming it with* ***as***). Named exports should always be enclosed within **{ }**.

```
1  export let myExportedVal = 10;
2  export function myFunc() { console.log('hello'); }
```

Note: ***export { }*** does not export an empty object. It is a no-op declaration that exports nothing.

Named export can be renamed while exporting from the module

The ***as*** operator can be used to rename the export declaration.

```
1  const func = () => { console.log('hello'); };
2  export { func as myFunction };
```

*The import declaration will have to use **myFunction** in order to access it.*

Exported members can also be **renamed** to something that's ***not a valid identifier***.

```
1  const func = () => { console.log('hello'); };
2  export { func as 'my-func' };
```

*The import declaration will have to use **"my-func"** in order to access it.*

The export declarations are not subject to **temporal dead zone** rules. Export declaration can be written, even before declaring the exported members.

Below code would work because the *export* statement is only a declaration, it does not utilize/evaluate the exported member.

```
1  export { x };
2  let x = 10;
```

*The export declaration of **"x"** is written even before declaring **"x"**.*

Default Export

There can be only one default export per module. They can be imported with any name. Default members are imported without curly brackets ({}).

```
// './exported-module.js'
export default let x = 10;
export default function greet() {
  console.log('Hello!');
}
```

Note: If multiple default exports are written in a module, each default export will **overwrite** the previous one. In the above example, **"greet()"** will be the default exported member.

```
// './main-program.js'
import greet from './exported-module.js';
greet(); // 'Hello!'
```

Note: The export default syntax allows any expression. For example: ***export default 10 + 20;***

Re-exporting/Aggregation

A module can also export the members which are exported from other modules without writing two separate *import/export* statements. This concept is called *Re-exporting* or *Aggregation*.

The ***from*** keyword is used with export declaration.

```
export { sayHi, sayBye }
       from './exportedModule.js';
```

We'll have to write two different *import/export* statements for the same purpose, without aggregation.

```
import { sayHi, sayBye }
       from './exportedModule.js';
export { sayHi, sayBye};
```

Wild-card (*) export statement

Wild card *export* statement re-exports all the **named exports** of a given *module*, as the *named exports* of the current *module*. But the *default export* of the given module is not re-exported.

Syntax
*export * from "<module-path>.js"*

```
export * from './exportedModule.js';
```

If there are two or more wild-card export statements that implicitly *re-exports* the same name, **neither one is re-exported**.

```
// 'module-1.js'
export const a = 10;

// 'module-2.js'
export const a = 10;

// 'module-aggregator.js'
export * from './module-1.js';
export * from './module-2.js';

// 'main-program.js'
import * as md from './module-aggregator.js';
console.log(md.a); // undefined
```

Aggregator re-exported the same member "a", therefore, it was not exported.

import statement

The *import* declaration is used to import members which are exported from other modules. The imported members are **read-only** with **live bindings**, because the importing module cannot modify them, they are only updated by the source module which exported them.

In order to use the *import* declaration, the file must be interpreted by the runtime as a *module*. In HTML, this is done by adding the ***type="module"*** attribute to the ***<script>*** tag. There is also a function-like dynamic *import()*, which does not require scripts of **type="module"** attribute.

There are four forms of import declaration:

- **Named import**
- **Default import**
- **Namespace import**
- **Side-effect import**

Named import

As the name suggests, this is used to import the *named export* members.

```
import { sayHi, sayBye }
        from './exportedModule.js';
```

Default import

As the name suggests, this is used to import the *default export* members.

```
import greet from './exportedModule.js';
```

Importing a name called **default** has the same effect as a *default import*. It is necessary to alias the name because *default* is a reserved word.

```
import { default as myDefaultMember}
        from './module-1.js';
```

Namespace import

Namespace import, inserts a given module into the current scope, containing all the exports from the module. All keys are *enumerable* in lexicographic order, with the ***default export*** available as a key called **default**.

```
import * as myModule from './exportedModule.js';
myModule.greet(); // 'Hello!'
```

*"**myModule**" is a **sealed** object with a null **prototype**.*

It is also possible to specify a default import with the namespace or named import. In such cases, the default import has to be declared before namespace/named import.

Example of default import with namespace import:

```
import defaultMember,
       * as myModule from './exported-module.js';
```

Example of default import with named import:

```
import defaultMember, { myExportedVal, myFunc}
       from './exported-module.js';
```

Note: JavaScript do not have **wild-card import**, like:
*import * from "exported-module.js"*,
similar to wild-card *export like:*
*export * from "module.js"*.

The reason is the high possibility of name conflicts.

Side-effect import

This does not import anything from the module, instead the module is imported for side-effects. As a side-effect, this import declaration executes the imported module's code.

This is often used for *polyfills*, which mutate the global variables.

Syntax
import "<path-of-the-module>";

```
1  import './module.js';
```

Summary

- **Modules** are used to split code and *import* when needed.
- The ***export*** uses a **"live binding"** mechanism for updating the references.
- To use an *export/import* statement, code must be interpreted as a **module** at runtime. In HTML, it is done by adding **type="module"** to the <*script*> tag.
- Modules are automatically interpreted in **strict mode**.
- **Named exports** have a unique name. A module can have multiple *named exports*. The ***"as"*** can be used to rename them while importing.
- There can be only **one default export** in a module.
- Re-exporting/Aggregation can be used to merge exports.
- The **wild-card (*)** *export* statement, re-exports all *named* exports.
- The ***import*** declaration is used to *import* the exported members.
- A module can have *named*, *default*, *namespace* and *side-effect* imports.
- The **side-effect** import does not *import* anything, instead it **executes** the imported module's code.

CHAPTER 13

‘this’ Keyword

Preface to 'this'

Introduction

Value of 'this' is the property of execution context (Global, function or eval)

Value of this in Global Context

Context outside of any function, window object in browser, globalThis

Value of this in Function Context

Depends on how function is called, call(), bind(), apply()

Value of this in Class Context

It is regular object with non-static members, constructor(), new()

Value of this in Inline Event Handler Context

this is set to DOM elements

The "this" keyword

In most cases, the value of ***this*** is determined by how a function is called (runtime binding). It can't be set by the assignment during execution and it may be different each time the function is called.

Value of "this"

The value of *this* is the property of an execution context (*Global*, *function* or *eval*), that is always a reference to an ***object*** in *non-strict mode* and can be ***any value*** in *strict mode*.

Value of "this" in the Global Context

In the global execution context (outside of any function), *this* refers to the global object, whether in strict mode or not. In the browser, *this* refers to the ***window*** object.

```
1  console.log(this === window); // true
```

*The variable defined **without** the **var**, **let** or **const** is by default defined in the global scope.*

```
1  a = 100;
2  console.log(window.a); // 100
```

Note: ***globalThis*** can be used to get a global object, regardless of current context in which code is running. For example, whether code is running in the browser or nodejs, *globalThis* will return global context.

Value of "this" in the Function Context

Inside a function, the value of *this* depends on how the function is called.

When "this" is not set by the function call

In Non-Strict Mode: In non-strict mode, *this* will default to the *global object*, which is a ***window*** in the browser.

```
1  function getThis() {  return this; }
2  console.log(getThis() === window); // true
```

In Strict Mode: In strict mode, the value of *this* will default to ***undefined***.

```
1  function getThisInStrictMode() {
2    'use strict';
3    return this;
4  }
5
6  console.log(getThisInStrictMode() === undefined); // true
```

When "this" is set by the function call

The ***call()***, ***apply()*** and ***bind()*** methods can be used to set the value of ***"this"*** for the called function.

```
1  const obj = { name: 'Name from obj' };
2  var name = 'Name from Global';
```

*The variable declared with the var keyword is added to the global object by default. Below code returns the global variable 'name', because **this** was not set by the function call, therefore **this** refers to a global **object**.*

```
function getName() {  return this.name; }
const nameValue = getName();
console.log(nameValue); // 'Name from Global'
```

*Below code returns the 'name' from the given object 'obj', because **call()** methods set the value of **this** to **obj**.*

```
function getName() {  return this.name; }
const nameValue = getName.call(obj);
console.log(nameValue); // 'Name from obj'
```

*Similarly, the below code returns the 'name' from the given object 'obj', because **apply()** methods set the value of **this** to **obj**.*

```
function getName() {  return this.name; }
const nameValue = getName.apply(obj);
console.log(nameValue); // 'Name from obj'
```

In **non-strict** mode, with ***call***, ***apply*** and ***bind***, if the value passed as *this,* is not an *object*, an attempt will be made to convert it to *object*. Values ***null*** and ***undefined*** becomes the *global object*.

```
function getThis() {
  console.log(Object.prototype.toString.call(this));
}

getThis.call(10); // [Object Number]
// Primitive number 10 is converted to object
// by new Number(10).

getThis.call(undefined); // [Object global]
// The undefined is converted to the global object.
```

*Let's understand **call()**, **apply()** and **bind()** methods in more detail.*

call()

This method calls the function with given ***this*** value and *arguments* provided individually.

Syntax

call()
this will be replaced with ***global*** in *non-strict mode* and *undefined* in *strict mode*.

*call(**thisArg**)*
this will be replaced with ***thisArg***.

*call(**thisArg**, arg1, arg2, ..., argN)*
***this** will be replaced with **thisArg** and following arguments will be assigned to function arguments.*

```
function sum(num1, num2) {
  return this.a + this.b + num1 + num2;
}
```

When variables are declared in the global object.

```
var a = 100;
var b = 200;

const res = sum.call(null, 2, 5);
console.log(res); // 307
```

*Because **null** was passed as the value of **thisArg**, **this** was replaced with the global object.*

Therefore: 100 + 200 + 2 + 5 = 307
this.a** **this.b** **num1** **num2

Let's see another example, where an object is passed to the call() method.

```
const obj = { a: 10, b: 20 };
const res = sum.call(obj, 5, 25);
console.log(res); // 60
```

*Because **obj** was passed as the value of **thisArg**, **this** was replaced with obj.*

Therefore: 10 + 20 + 5 + 25 = 60
this.a** **this.b** **num1** **num2

The ***call()*** allows for a function/method belonging to one *object* to be assigned and called for a different *object*.

```
const obj = {
  a: 10,
  b: 20,
  sum(num1, num2) {
    return this.a + this.b + num1 + num2;
  }
};

const anotherObj = { a: 100, b: 200 };
const res = obj.sum.call(anotherObj, 5, 5);
console.log(res); // 307
```

apply()

This is similar to the *call()* method, with just one difference that *arguments* are passed as an *array*.

Syntax

apply()
this will be replaced with ***global*** in *non-strict mode* and undefined in *strict mode*.

*apply(**thisArg**)*
this will be replaced with ***thisArg***.

*apply(**thisArg**, [arg1, arg2, ..., argN])*
***this** will be replaced with **thisArg** and the following **array of arguments** will be assigned to function arguments*.

```
const obj = { a: 10, b: 20 };
const res = sum.apply(obj, [5, 25]);
console.log(res); // 60
```

*Because **obj** was passed as the value of **thisArg**, **this** was replaced with **obj** and **[5, 25]** were assigned to function parameters.*

Therefore: 10 + 20 + 5 + 25 = 60
this.a this.b num1 num2

bind()

This method creates a new *function* that, when called, has it's *this*, set to the provided value, with the given arguments preceding any arguments provided when the new function is called.

Syntax

*bind(**thisArg**)*
this will be replaced with ***thisArg***.

*bind(**thisArg**, arg1, arg2, ..., argN)*
***this** will be replaced with **thisArg** and following arguments will be assigned to function arguments*.

bind() is similar to ***call()***, with a difference that ***thisArg*** and ***arguments*** can't be reassigned i.e., it holds the values from the **first bind call**.

The *bind()* method returns a copy of the given function with the specified ***this*** value and initial arguments (if provided).
The *bind()* function creates a *new bound function*. Calling a *bound function* generally results in the execution of its wrapper function.

The bound function will store the parameters passed, which includes the value of *this* and the arguments, as its

internal state. These values are stored in advance, instead of being passed at the time of calling the function.

This concept is also called **Function Currying**, where a function is created from an existing function, by presetting some of the parameters.

```
function multiply(a, b) { return a * b; }
```

```
const multiplyByTwo = multiply.bind(this, 2);
```

*The argument **"a"** of the multiply() function is assigned the value **2** in advance.*

```
const multiplyByThree = multiply.bind(this, 3);
```

*Similarly, argument **"a"** of the multiply() function is assigned the value **3** in advance.*

```
function multiply(a, b) {
  return a * b;
}

const multiplyByTwo = multiply.bind(this, 2);
console.log(multiplyByTwo(5)); // 10

const multiplyByThree = multiply.bind(this, 3);
console.log(multiplyByThree(10)); // 30
```

Below code demonstrates that binding can't be reassigned.

```
function sum(num1, num2) {
  return this.a + this.b + num1 + num2;
}

const obj = { a: 10, b: 20 };
const newSumFn = sum.bind(obj, 5, 15);
const res = newSumFn();
console.log(res); // 50
```

Because: 10 + 20 + 5 + 15 = 50
this.a this.b num1 num2

```
const res = newSumFn(100, 200);
console.log(res); // 50
```

This proves that arguments can't be reassigned. Values given at the time of calling function are ignored here.

*Below code proves that **this** can't be reassigned. Values given at the time of calling function are ignored here. Also, observe that **anotherSumFn** was created from an earlier binded function **newSumFn** and not **sum**.*

```
const anotherObj = { a: 100, b: 200 };
const anotherSumFn = newSumFn(anotherObj, 500, 1000);
const res2 = anotherSumFn();
console.log(res2); // 50
```

Note: Original *function* can be bound multiple times.
*const anotherSumFn = **sum.bind**(anotherObj, 500, 1000);*
const res2 = anotherSumFn();
*console.log(res2); // **1800***
Because: 100 + 200 + 500 + 100 = 1800
this.a this.b num1 num2

Value of "this" in the Class Context

The behavior of ***this*** in *class* and *function* is similar, since *classes* are *functions* under the hood.

Within a *class constructor*, this is a regular *object*. All *non-static methods* of the class are added to the *prototype* of *this*.

Note: static methods are not properties of ***this***, they are properties of the *class* itself.

```
class User {
  constructor() {
    const proto = Object.getPrototypeOf(this);
    console.log(Object.getOwnPropertyNames(proto));
  }

  firstName() {
    console.log('firstName instance method');
  }
  lastName() {
    console.log('lastName instance method');
  }
  static address() {
    console.log('address static method');
  }
}
```

Object.getPrototypeOf()

This method returns the *prototype* of the given *object*, in our case a prototype of ***this***.

Object.getOwnPropertyNames()

This method returns the *array* of property names, which are the own properties of the given object (excluding the prototype properties).

```
new User(); // ['constructor', 'firstName', 'lastName']
```

address() *is not returned in array, because* ***static*** *methods are properties of class and not the instance (****this****).*

Value of "this" in Derived Classes

Unlike base class constructors, derived class constructors have no initial *this* binding. Calling ***super()*** creates *this* binding within the constructor.

The *super()* call is eventually equivalent to: ***this*** **= new Base()**.

Derived classes must not return before calling *super()*, unless they return an ***Object*** or have *no constructor*.

Referring to ***this*** before calling *super()* in the derived class will throw an error.

```
1  class Base {}
2  class GoodDerived extends Base {}
3  new GoodDerived();
```

*It will work, because the derived class does not have a **constructor**.*

*Below code will work, because the derived class constructor returns an object, therefore, **this** will be initialized with **{ a: 5 }**.*

```
1  class Base {}
2  class AlsoGoodDerived extends Base {
3    constructor() {
4      return { a: 5 };
5    }
6  }
7
8  new AlsoGoodDerived();
```

*Below code will throw an error, because the derived class has a constructor, but neither it calls **super()**, nor it returns an **object**.*

```
1  class Base {}
2  class BadDerived extends Base {
3    constructor() {}
4  }
5
6  new BadDerived();
```

Note: Classes are always **strict mode** code, i.e, calling methods with an **undefined this** will throw an error.

The "new" Operator

The ***new*** operator creates an instance of a user-defined object type or of one of the built in object types that has a ***constructor function***. For example: The ***Date*** object.

Syntax

new ***constructor****[([****arguments****])]*

constructor: A *class* or *function* that specifies the type of *object* instance.

arguments: A list of values that the ***constructor*** will be called with.

Using "new" with Functions

When a *function* is called with the ***new*** operator, the *function* will be used as a *constructor*. The ***new*** operator will do following things:

1. Creates a blank *object*.
   For convenience let's call it the **newInstance**.

2. Points the **newInstance**'s *[[Prototype]]* to the constructor function's ***prototype*** property.

3. Executes the *constructor* function with given *arguments*, binding **newInstance** as ***this*** context.

4. If the constructor function returns a ***non-primitive value***, the returned value becomes the result of the ***new*** expression. Otherwise, if the *constructor* function doesn't return anything or returns a ***primitive value***, **newInstance** is returned instead.

Normally constructors don't return a value, but they can choose to do so to override the object creation process.

*In the below code, the **new** operator binds **this** with the **created instance**.*

```
function User(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.showName = function () {
    console.log(this.firstName + '-' + this.lastName);
  };
}

const user = new User('JavaScript', 'Language');
user.showName(); // 'JavaScript-Language'
```

*Below code will throw an error, because the **return** statement **overrides the new response** with the returned **object {a: 5}** and the returned object does not contain any function named "**showName()**".*

```
function User(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.showName = function () {
    console.log(this.firstName + '-' + this.lastName);
  };
  return { a: 5 };
}

const user = new User('JavaScript', 'Language');
console.log(user); // {a: 5}

user.showName(); // 'Error usr.showName is not a function'
```

new.target

A *function* can know whether it is invoked with *new* by checking the ***"new.target"*** property. The *"new.target"* is ***undefined*** when the function is invoked ***without a new operator***.

Syntax

new.target

Return Value:

- When called **with new**, it returns reference to the *constructor* or *function*.
- When called **without new**, it returns *undefined*.

```
function User(firstName, lastName) {
  if (!new.target) {
    throw 'User() must be called with new operator';
  }
  this.firstName = firstName;
  this.lastName = lastName;
  this.showName = function () {
    console.log(this.firstName + '-' + this.lastName);
  };
}
```

throw *is used to throw/return error information.*

```
const user = User('JavaScript', 'Language');
// 'Error: User() must be called with new operator'
// Because User() was called without a new operator.

const user1 = new User('JavaScript', 'Language');
console.log(user1.showName());
// 'JavaScript-Language'
```

Prior to ES6, which introduced the ***class***, most JavaScript built-ins were both *callable* and *constructible*, although many of them exhibit different behaviors.

For example:

1. ***Array()***, ***Error()*** and ***Function()*** behave the same way, when called as a *function* or *constructor*.

2. ***Boolean()***, ***Number()*** and ***String()*** *coerce* their arguments to the respective *primitive type* when called as a *function*, but returns *wrapper objects* when *constructed* with the ***new*** operator.

But after ES6, language is stricter about which are *constructors* and which are *functions*.

For example:

1. ***Symbol()*** and ***BigInt()*** can only be called without ***new***. Attempting to construct then will throw **TypeError**.

2. ***Proxy*** and ***Map*** can only be constructed with ***new***. Attempting to call them as functions will throw a **TypeError**.

Using "new" with Classes

Classes can only be instantiated with the ***new*** operator. Attempting to call a class without *new* will throw **TypeError**.

```
class User {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  showName() {
    console.log(this.firstName + '-' + this.lastName);
  }
}

const user = new User('JavaScript', 'Language');
user.showName(); // 'JavaScript-Language'
```

*Because the **new** operator binds **this** with the **created instance**.*

Value of "this" in the Inline Event Handler

When code is called from an ***inline on-event handler***, it's ***this*** is set to the **DOM element**, on which the listener is placed.

```
<button onclick='console.log(this.tagName)'>Button</button>;
```

*When the user clicks on the button, it logs **"button"**, which shows that **this** here is pointing to the DOM element (button).*

Summary

- The value of ***this*** is the property of an execution context (*Global, function or eval*).
- In the browser, ***this*** in **global** context refers to the ***window*** object.
- The ***call()***, ***apply()*** and ***bind()*** methods can be used to set ***this*** for *function context*.
- When a function is created from an existing function by presetting parameters, it is called **function currying**.
- In the *class constructor*, ***this*** is a *regular object* with all *non-static* members.
- The **new** operator creates an instance of a *user defined object* that has a *constructor function*.
- ***Classes*** can only be instantiated with the ***new*** operator.
- The ***"new.target"*** tells whether a function is called with a ***new*** operator or not.
- ***Symbol()*** and ***BigInt()*** can only be called without a ***new*** operator.
- With event handlers, ***this*** is set to the **DOM** element.

CHAPTER 14

Class

Preface to Class

Introduction

Class Declaration, Class Expression, class definition is not hoisted, constructor(), static members, binding 'this'

Private Fields

Declared with '#' prefix, can't be inherited, can't be accessed outside, can't declare and delete in constructor()

Inheritance

Base/Parent/Inherited classes, Child/Derived classes, MIX-INS

Class

The classes are templates for creating objects. They encapsulate data with methods to work on that data. Classes in fact are special functions.

In JavaScript, classes are mainly an abstraction over the existing prototypical inheritance mechanism. Classes themselves are normal JavaScript values as well, and have their own prototype chain.

Below are the key features of classes:

- Constructor
- Instance methods and fields
- Static methods and fields

Classes can be defined in two ways:

1. **Class Declaration**
2. **Class Expression**

Class Declaration

The class declaration creates a binding of a new *class* to a given name. Class declarations are scoped to blocks as well as functions.

```
class User {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  showName() {
    console.log(this.firstName + '-' + this.lastName);
  }
}
```

Class Expression

Class expressions can be **named** or **unnamed**. *Below class is an example of named class expression*:

```
const User = class User {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  showName() {
    console.log(this.firstName + '-' + this.lastName);
  }
};
```

Below class is an example of unnamed class expression:

```
const User = class {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  showName() {
    console.log(this.firstName + '-' + this.lastName);
  }
};
```

Class definition is ***not hoisted***, which means *class* must be defined before they are constructed with the ***new*** operator.

```
const usr = new User('JavaScript', 'Language');
// Reference Error

class User {  /*.......... */ }
```

The body of a class is executed in ***strict mode***, therefore, all restrictions of *strict mode* are applicable to the ***class*** body.

constructor

The *constructor* method is a special method for creating and initializing an object created with a *class*.

There are some syntax restrictions:

- Constructor cannot be a *getter*, *setter*, *async* or *generator*.
- A *class* cannot have more than one *constructor* method.

A *constructor* can use ***super*** keyword to call the *constructor* of the base/super class.

```
class User {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
}
```

static Methods and Properties

The ***static*** keyword defines a *static method* or *property* for a *class*. Static members are called without instantiating the *class* and cannot be called through a *class* instance.

The *static* methods are often used to create utility functions for the *class*, whereas *static* properties are useful for caches, fixed-configuration or any other data that don't need to be replicated across *class* instances.

```
class User {
  // 'MINIMUM_AGE' is a static property
  static MINIMUM_AGE = 18;
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  getUserDetail() {
    return this.name + ' - ' + this.age;
  }

  // 'getValidUsers()' is a static method
  static getValidUsers(userList) {
    return userList.filter((usr) =>
              usr.age > this.MINIMUM_AGE);
  }
}
```

*In the code below, the getUserDetail() method can be called by the class instance variable "**usr**".*

```
const usr = new User('John', 25);
console.log(usr.getUserDetail()); // 'John - 25'
```

In the code below, ***getValidUsers()*** *is called on the* ***User class*** *and not on the instance of the class, like* ***usr***.

```
const usr2 = new User('Jack', 17);
const validUsers = User.getValidUsers([usr, usr2]);
console.log(validUsers); // [{ name : 'John', age : 25}]
```

But, below code will print undefined, because static members can't be accessed by instance variables.

```
console.log(usr2.MINIMUM_AGE); // undefined
```

Binding "this" with prototype and static methods

When a *static* or *prototype/instance* methods are called without a value of ***this***, the value of *this* will be ***undefined*** inside the method.

*Below code will throw an error, because **this** was **undefined** in the method.*

```
const usr = new User('John', 25);
const getUserInfo = usr.getUserDetail();

getUserInfo();
// 'TypeError: Cannot read properties of undefined'
```

Private field declarations

Private class members can be created by using **"#"** prefix. The privacy encapsulation of these class features is enforced by JavaScript itself. The **#** is a part of the member name.

Private fields can't be neither declared nor deleted inside the constructor. It is a syntax error to access private members outside the class. Also private members can't be inherited.

Private instance/prototype members

- Can only be accessed inside the class.
- Can't be inherited

Private static members

- Are only accessible on the class itself or in ***this*** context of static members.
- Can't be inherited.

Let's create a class with private members.

```
class User {
  // '#GROUP' is a private property
  #GROUP = 'Student';
  static MINIMUM_AGE = 18;
  constructor(name, age) {
    this.name = name;
    this.age = age;

    delete this.#GROUP;
    this.#privatePropInConstructor = 10;
    // If we try to delete or declare new private
    // properties inside the constructor,
    // it'll throw an error.
  }

  // '#getUserDetail()' is a private method
  #getUserDetail() {
    return this.name + ' - ' + this.age;
  }

  getUserInfo() {
      return this.#GROUP + ':' + this.#getUserDetail();
  }
}
```

Below code will work fine because we called the public method.

```
const usr = new User('John', 25);
console.log(usr.getUserInfo()); // 'Student: John - 25'
```

*But, below code will throw an error: "**SyntaxError: Private field #GROUP must be declared in an enclosing class**", because private members can't be called outside class.*

```
console.log(User.#GROUP);
```

Inheritance

The ***extends*** *keyword* is used to create a *class* as a child of another *class*, this mechanism is called **inheritance**.

If there's a *constructor* present in the subclass, it needs to first call *super()* before using *this*.

Note: The *super* keyword can also be used to call corresponding methods of super class.

Base/Parent/Inherited Class

This is the *class* which will be used as a base for creating child *classes*. The basic role of ***base class*** is to contain *common functionalities*, which can be used by ***multiple child classes***.

```
class Vehicle {
  constructor(vehicleType, make, wheels) {
    this.vehicleType = vehicleType;
    this.make = make;
    this.wheels = wheels;
  }

  showVehicleInfo() {
    console.log(`${this.make} ${this.vehicleType}
                have ${this.wheels} wheels`);
  }
}
```

Child/Derived Class

This is the *class* which ***extends*** the functionality of the *base class*. The role of a *derived class* is to perform more specialized tasks. Derived *class* **uses/overrides** the base class members, based on the need.

super(): Calling ***super()*** is important. It calls the parent class *constructor* with the list of passed *arguments*. The ***super*** can also be used to call **public** *properties/methods* of the parent class.

*In code below, **showVehicleInfo()** is not defined in **Bike class**, but still this works, since it was **inherited** from the base class **Vehicle**.*

```
class Bike extends Vehicle {
  constructor(make) {
    super('Bike', make, 2);
  }
}

const bike = new Bike('Honda');
bike.showVehicleInfo(); // 'Honda Bike have 2 wheels'
```

*Similarly, in the below code, **showVehicleInfo()** is not defined in **Car class**, but still this works, because it was **inherited** from the base class **Vehicle**.*

```
class Car extends Vehicle {
  constructor(make) {
    super('Car', make, 4);
  }
}

const car = new Car('Maruti');
car.showVehicleInfo(); // 'Maruti Car have 4 wheels'
```

MIX-INS

Abstract classes or **mix-ins** are templates for classes. Class inheritance has a limitation that a *class* can only extend one *class*.

A function with a super class as input and a subclass extending that superclass as output can be used to implement *mix-ins* in JavaScript.

*Let's understand MIX-INS with an example. In the code below, the "**BaseClass**" will be the base class of the derived classes.*

```
class BaseClass {
  constructor(numArr) { this.numArr = numArr;  }
}
```

*The **sumMixin()** is a function that accepts a base class and extends it to return the sum of a given number array.*

```
const sumMixin = (Base) =>
  class extends Base {
    constructor(numArr) {
      super(numArr);
    }
    sum() {
      return this.numArr.reduce((a, b) => a + b);
    }
  };
```

*Similarly, **averageMixin()** is a function that accepts a base class and extends it to return the average of a given number array.*

```
const averageMixin = (Base) =>
  class extends Base {
    constructor(numArr) {
      super(numArr);
    }
    average() {
      return this.numArr.reduce((a, b) => a + b)
             / this.numArr.length;
    }
  };
```

Now, let's create a derived class extending the above defined mixin methods.

```
class MathClass extends sumMixin(averageMixin(BaseClass)) {}

const math = new MathClass([2, 5, 6, 8]);
console.log(math.sum()); // 21
console.log(math.average()); // 5.25
```

Summary

- **Classes** are templates for *objects*, encapsulating data and methods.
- Classes can be defined as: **Class Declaration** and **Class Expression**.
- Class definition is not **hoisted**. The body of class executes in **strict mode**.
- ***Constructor*** is a special method for creating and initializing class instances.
- **Static members** are called without instantiating class and can't be called with *class instances*.
- When a ***static*** or *prototype/instance* methods are called without a value of ***this***, the value of ***this*** will be ***undefined*** inside the method.
- **Private** members are creating using the **"#"** prefix.
- The ***extends*** keyword is used to implement **inheritance**.
- The ***super()*** calls the parent constructor with given parameters.
- **Abstract classes** or **mix-ins** are templates for classes, used to ***extend*** a class from multiple parent classes.

CHAPTER 15

Inheritance & Prototype

Preface to Inheritance & Prototype

Introduction

Inheritance implemented by prototype, null representing the end of chain

Setting Prototype Values

Set prototype via Object.setPrototypeOf(), Object.create(), class

Inheritance and the Prototype chain

When it comes to *inheritance*, JavaScript has only one construct: ***Objects***. Each *object* has a private property which holds a link to another object called its ***prototype***.

The *prototype* object has a *prototype* of its own, and so on until an object is reached with **null** as its *prototype*.

By definition, **null** has no *prototype* and acts as the final link in the **prototype chain**. **Classes** also use the same inheritance mechanism under the hood.

Inheriting Properties

JavaScript *objects* are dynamic **"bags"** of properties, which are referred to as their **own properties**. When we try to access a property of an object, the property will not just be searched on the object but will also be searched on the prototype *chain*, until either property is found or end of prototype chain is reached.

```
const obj = {
  name: 'JavaScript',
  __proto__: {
    version: 6
  }
};
```

*In the code below, **name** property was found in the **obj** object.*

```
console.log(obj.name); // 'JavaScript'
```

```
console.log(obj.version); // 6
```

*The reason for **"6"** being printed as **"obj.version"** is the **recursive search** in the **obj** and the prototype chain.*

1. *It searched for "version" in its **own properties** of **"obj"**, **Not found**.*
2. *Then, it searches the prototype chain and finds the "version" in the __proto__ object and returns **"6"**.*

Now, let's try to access a non-existing property.

```
console.log(obj.year); // undefined
```

*The reason for **undefined**, can be traced with the below steps:*

1. *Searches "year" in obj's own properties, **Not Found**.*
2. *Searches "year" in obj's prototype chain object, **Not Found**.*
3. ***End of prototype chain reached**.*
4. *Returns **undefined**.*

__proto__

It is one of the ways to **set the prototype of an object**.

There are other ways to set the prototype:

- By using **Objet.setPrototypeOf()**
- By using **Object.create()**
- By creating **class**

Let's see how properties/methods can be inherited using ***__proto__****.*

```
const parent = {
  label: 'Parent Label',
  show() {
    console.log(this.label);
  }
};

const child = { __proto__: parent };
```

```
parent.show(); // 'Parent Label'
```

When parent.show() is called, ***this*** *refers to the parent object.*

```
child.show(); // 'Parent Label'
```

When child.show() is called, ***this*** *refers to the child object. The property* ***"label"*** *is inherited from the* ***"parent"*** *object.*

```
child.label = 'Child Label';
child.show(); // 'Child Label'
```

Shadows *the "label" property on the parent by adding "label" property to the own properties of the child object.*

Prototypes with Classes

The power of prototypes is that we can reuse a set of properties if they should be present on every instance.

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

User.prototype.showDetails = function () {
  console.log(`Username: ${this.name}`);
};
```

*The above statement will add the **showDetail()** method to every instance created from constructor function **User()**. This is because every instance created from a constructor function will automatically have the constructor's prototype property as it's **[[Prototype]]**.*

```
const users = [new User('John'), new User('Jack')];
users.map((usr) => usr.showDetails());
// Output:
// 'Username: John'
// 'Username: Jack'
```

Let's implement the same using the classes.

Classes are syntax sugar over constructor functions, which means we can still manipulate the class *prototype* to change the behavior of all the instances.

```
class User {
  constructor(name) {
    this.name = name;
  }
  showDetail() {
    console.log(`Username: ${this.name}`);
  }
}
```

Note: Because ***prototype*** references the same *object* as the *[[Prototype]]* of all instances, we can change the behavior of all instances by **mutating the prototype** of the constructor function.

Note: Extending any **built-in prototype** can be a **misfeature**. For example, defining ***Array.prototype.myMethod = function () { /*.....*/ }*** and then using myMethod() on all array instances.

This misfeature is called **"Monkey Patching"**. Doing monkey patching risks forward compatibility, because if JavaScript, in future, adds **"myMethod()"** but with a different signature, it will break the code.

Due to historical reasons, some *built-in* constructors *prototype* properties are *instances* themselves.

Number.prototype is a number 0 (zero).

```
Number.prototype + 1; // 1
```

Similarly, below prototypes are instances themselves.

```
1  Array.prototype.map((x) => x + 1); // []
2  String.prototype + 'a'; // 'a'
```

Pros of using *prototype*

- **Fast**, **standard** and ***JIT optimizable*** method of setting prototype.
- High **readability** and **maintainability**.

Cons of using *prototype*

- Classes, especially with private properties are ***less optimized***.
- Not supported in older environments and **may need transpilers**.

Object.setPrototypeOf()

This method can be used to set the *[[Prototype]]* of the *constructor.prototype*.

Syntax
*Object.**setPrototypeOf**(<destination-prototype>, <source-prototype>)*

```
1  class BaseClass {  /*....*/ }
2  class DerivedClass {  /*....*/ }
3  Object.setPrototypeOf(DerivedClass.prototype,
4                        BaseClass.prototype);
```

*Above code sets the BaseClass prototype to DerivedClass prototype. This is equivalent to **extends** of **class**.*

Pros of using *setPrototypeOf()*

- Allows the *dynamic manipulation* and can even *force a prototype* on a **prototype-less** object created using *Object.create(null)*.

Cons of using *setPrototypeOf()*

- ***Ill performing***, setting prototypes dynamically disrupts the optimization process.
- Might cause some JavaScript Engines to ***recompile*** the code for *de-optimization*.

Object.create()

This method ***reassigns*** the *prototype* property, and for this reason it is considered as **bad practice**.

```
DerivedClass.prototype = Object.create(
  BaseClass.prototype
);
```

Pros of using *Object.create()*

- Allows **directly setting** *[[Prototype]]* of an object at the creation time, which permits the runtime to further optimize.
- Allows creation of objects without a prototype, using ***Object.create(null)***.

Cons of using *Object.create()*

- It can slow down object initialization, which **may cause performance** issues.

Summary

- Each *object* has a private property called ***"__proto__"*** called its **prototype**.
- The ***null*** value of ***__proto__*** indicates the end of the prototype chain.
- The *prototype* can be set by: ***Object.setPrototypeOf()***, ***Object.create()*** and by ***class***.
- Some *built-in constructors prototype* properties are **instances themselves**. For example: ***Number.prototype*** is a number **0** (zero).
- ***Object.setPrototypeOf()*** is equivalent to ***extends*** of *class*, allows dynamic manipulation, and has negative impact on performance.
- ***Object.create()*** **reassigns** the *prototype*, allows directly setting *[[Prototype]]*, allows creating objects **without** *prototype*, may cause performance issues.

CHAPTER 16

Strict Mode

Preface to Strict Mode

Introduction

Introduced in ES5, is a restricted variant of JavaScript, can co-exist with non-strict mode

Invoking Strict Mode

'use strict' can be applied to entire script or individual function

Applied Restrictions

Restrictions are applied on classes, modules, variables, eval(), arguments, etc

Strict mode

JavaScript's ***strict mode***, introduced in ES5, is a way to ***opt in*** to a restricted variant of JavaScript. The *strict mode* isn't just a subset, it intentionally has different semantics from normal code. The strict mode and non-strict mode code can co-exist.

Strict mode make several changes to normal JavaScript semantics:

1. Changes some silently ignored errors to be thrown, i.e, errors which are ignored in non-strict mode will throw errors in strict mode.
2. Fixes mistakes that make it difficult for JavaScript engines to perform optimizations, i.e, strict mode code can sometimes be made to run faster than non-strict mode code.
3. Prohibits some syntax likely to be defined in future versions of *ECMAScript*.

Invoking strict mode

Strict mode may be applied to **entire scripts** or to **individual functions**. It doesn't apply to *block statements* enclosed in { }, attempting to apply *strict mode* to *block statements* will do nothing.

The ***eval()*** code, ***Function*** code, ***event handle attributes***, strings passed to ***setTimeout()*** and related functions are considered as **entire scripts**, therefore, invoking *strict mode* in them works as expected.

Syntax
"use strict"

*Put the statement **"use strict"** as the first statement to make the **entire script** or **function** run in **strict mode**.*

```
1  'use strict';
2  const name = 'JavaScript, is in strict mode';
```

Note: In strict mode, starting with ES2015, functions inside blocks are scoped to that block. **Prior to ES2015, block-level functions were forbidden in strict mode**.

Restrictions applied by strict mode

Below are a few of the restrictions implemented by the strict mode.

- ***Class*** and ***JavaScript modules*** code automatically runs in strict mode.

- Strict mode makes it impossible to accidentally create **global** variables.

```
1  'use strict';
2  let misTypedVar;
3  mistypedvar = 10;
```

*Assuming no global variable **"mistypedvar"** exists, this will throw a **ReferenceError**, due to spelling mistake. (**Non-strict** mode would have declared a new global variable named **"mistypedvar"**).*

- Strict mode makes assignment to non-writable global variables throw an exception.

```
let undefined = 10; // throws TypeError
```

- In strict mode, attempting to delete undeletable properties throws errors.
- Strict mode requires that function **parameter names to be unique**.
- Strict mode in ES5, **forbids** a ***0-prefixed*** octal literals.
- Strict mode in ES6 forbids setting properties on primitive values.

```
(10).text = 'Ten'; // TypeError
```

- In ES5 strict mode code, duplicate property names were considered a SyntaxError. But with the introduction of **"computed property names"**, making duplication possible at run time, therefore, ES6 removed this restriction.
- Strict mode prohibits ***with*** keyword.
- ***eval()*** of strict mode code ***does not introduce new variables*** into the surrounding scope.
- ***eval*** and ***arguments*** can't be bound or assigned.

```
eval = 10; // SyntaxError
arguments++; // SyntaxError
```

- Strict mode code doesn't alias properties of ***arguments*** object.

  In non-strict mode code, a function whose argument is let say **"arg"**, then setting **"arg"** also sets ***arguments[0]*** and vice-versa.

```
function normalFunc(arg) {
  arg = 10;
  return [arg, arguments[0]];
}

const res = normalFunc(15);
console.log(res[0]); // 10
console.log(res[1]); // 10
// The value of arguments[0] was also changed.
```

  While in strict mode code, setting **"arg"** does not change ***arguments[0]***.

```
function strictFunc(arg) {
  'use strict';
  arg = 10;
  return [arg, arguments[0]];
}

const res = normalFunc(15);
console.log(res[0]); // 10

console.log(res[1]); // 15
// The value of arguments[0] was not changed.
```

- In strict mode, ***arguments.callee is no longer supported***. In non-strict mode code, argument.callee refers to the enclosing function.

Summary

- The ***strict mode***, introduced in ES5, is a way to opt-in to restricted JavaScript.
- The *strict mode* and *non-strict* code can co-exist.
- Strict mode changes silent errors to be **thrown**, fixes mistakes to improve performance, prohibits future syntaxes.
- Add ***"use strict"*** as the first statement to run code in strict mode, may be applied to the entire script or individual function, but not to block statements.
- *Class* and *Modules* code automatically runs in strict mode. Assignment to **non-writable** will throw errors.
- Strict mode restricts: deleting undeletable, function parameter names to be unique, adding duplicate properties, etc.

CHAPTER 17

Hoisting

Preface to Hoisting

Introduction

It is a process where declarations are moved to top of the scope

Function Hoisting

Allows functions to be safely, declarations are hoisted but definitions are not

Variable Hoisting

Variable declaration is hoisted but not the initialization

Class Hoisting

Class declaration is hoisted, not initialized by default

Hoisting

JavaScript **hoisting** refers to the process whereby the *interpreter* appears to move the declaration of *functions*, *variables* or *classes* to the top of their scope, prior to the execution of the code.

Any of the below behaviors may be regarded as hoisting:

1. Being able to use a variable's value before the line it is declared (**"Value hoisting"**).
2. Being able to reference a variable before the line it is declared, without throwing a *ReferenceError*, but the value is always *undefined* (**"Declaration hoisting"**).
3. The declaration of the variable causes behavior changes in its scope before the line in which it is declared.

Note: *Hoisting* is not a term defined in *ECMAScript* specifications. The spec define a group of declarations as **HoistableDeclaration**, but only includes *function*, *function**, *async function* and *async function** declarations.

Function Hoisting

Hoisting allows *functions* to be safely used in code before they are declared. **Function declarations** are hoisted, but **function expressions** are not hoisted.

```
greet(); // 'Function declarations are hoisted'
function greet() {
  console.log('Function declarations are hoisted');
}
```

*Below code throws **ReferenceError**, because **function expressions are not hoisted**.*

```
greet(); // ReferenceError
const greet = () => {
  console.log('Function expressions are NOT hoisted');
};
```

Variable Hoisting

Hoisting also works with *variables* too. Variables declared with *var* are hoisted.

However, JavaScript only **hoists *declaration*** and not the *initialization* i.e, the initialization doesn't happen until the associated line of code is executed, even if the variable was originally initialized and then declared or declared and initialized in the same line.

Until that point in the execution is reached, the variable has its default value (*undefined* for a variable declared with ***var***, otherwise uninitialized).

```
console.log(name); // undefined
var name = 'JavaScript';
```

If we forget the declaration altogether (and only initialize the value) the variable isn't hoisted. Trying to access the variable before it is initialized will result in **ReferenceError**.

```
1  console.log(num); // ReferenceError
2  num = 10;
```

*The above statement will throw **ReferenceError**, because num was not hoisted.*

Class Hoisting

Classes defined using a ***class declaration*** are hoisted, which means that JavaScript has a reference to the *class*.

However, class is not initialized by default, so any code that uses class before its initialization will throw a **ReferenceError**. On the other hand, **class expressions** *are **NOT hoisted***.

```
1  const user = new User('John'); // ReferenceError
2
3  class User {
4    constructor(name) {
5      this.name = name;
6    }
7    showDetail() {
8      console.log(`Username: ${this.name}`);
9    }
10 }
```

Summary

- In **hoisting**, declarations of *functions*, *variables* or *classes* are moved to **top** of their *scope*, prior to the execution of the code.
- **Function declarations** are hoisted but **function expressions** are not hoisted.
- JavaScript only hoists **declarations** and not the **initialization**.

CHAPTER 18

Exception Handling

Preface to Exception Handling

Introduction

Code exceptions can be handled by try...catch...finally statements

try Block

Contains statements that may throw error

catch Block

Contains statements that specify how to handle errors

finally Block

Contains statements to be executed after try and catch statements

Exception Handling

An exception is a condition that interrupts normal code execution. Code exceptions can be thrown using the ***throw*** statement and can be handled using ***try...catch...finally*** statements.

throw

A ***throw*** statement specifies the value to be thrown. Any value can be thrown like numbers, strings, objects or any other valid literal.

Execution of the current function will stop (the statements after *throw* won't be executed), and control will be passed to the first *catch* block. If no *catch* block exists, the program will terminate.

Syntax
throw *<value-to-throw>*

Example:
throw *"Error occurred in processing!"*

try block

The ***try*** block contains one or more statements which may throw any error.

Syntax

```
try {
        // statements
}
```

catch block

The ***catch*** block contains statements that specify what to do if an exception is thrown from the *try* block.

finally block

The ***finally*** block contains statements to be executed after the try and catch block executes. The *finally* block will execute whether or not an exception is thrown.

If the *finally* block returns a value, this value becomes the return value of the entire *try..catch..finally* block, overriding values returned from try or catch block, if any. Often used to perform cleanup tasks like closing connections, unsubscribing, etc.

```
try {
  throw 'Error thrown';
} catch (e) {
  console.log(e); // 'Error thrown'
  return false;
  // Return value will be ignored,
  // because finally block also returns a value.
} finally {
  console.log('Finally block executed');
  return true;
}

// Output:
// Error thrown
// Finally block executed
```

We can use ***Error()*** constructor for throwing errors. Error thrown using *Error() constructor* contains many properties, out of which we will use ***name*** and ***message*** properties to get more refined error messages.

The ***name*** property provides the general class of **Error**, while ***message*** provides a more succinct message.

```
try {
  throw new Error('Error thrown');
} catch (e) {
  console.log({ name: e.name, message: e.message });
  // { name: 'Error', message: 'Error thrown'}
} finally {
  console.log('Finally block executed');
  return true;
}

// Output:
// { name: 'Error', message: 'Error thrown'}
// Finally block executed
```

Summary

- The ***try***, ***catch*** and ***finally*** statements are used to handle *runtime exceptions*.
- The ***throw*** statement specifies the value to be thrown, can be any valid value like *number*, *string*, *object*, etc.
- The ***finally*** block contains statements to be executed after ***try*** and ***catch*** block executes. If a value is returned from the finally block, it becomes the return value of the entire *try...catch...finally* block.
- The ***Error()*** constructor throws more succinct messages. Contains **"name"**, **"message"**, **"cause"**, *etc* with error details.

CHAPTER 19

Iterators & Generators

Preface to Iterators & Generators

Introduction

This brings the concept of iteration, customizing loop behavior

Iterators

Object implementing iterator protocol, next(), yield

Generators

*Function to define iterative algorithm, execution is not continuous, function**

Iterables

Objects which define iteration behavior, @@iterator, return(value)

Iterators and Generators

Iterators and generators bring the concept of iteration directly into core language and provide a mechanism for customizing the behavior of *for...of* loop.

Iterators

An iterator is any *object* which implements the iterator protocol by having a *next()* method that returns an *object* with two properties:

1. **value**
   The next value in the iteration sequence.

2. **done**
   This is ***true*** if the last value in the sequence has already been consumed. If ***value*** is present with ***done***, it is the iterator's return value.

Once created, an iterator object can be iterated explicitly by repeatedly calling ***next()*** method. Iterating over an iterator is said to consume the iterator, because it is generally only possible to do once.

After a terminating value has been **yielded**, additional calls to *next()* should continue to ***return { done: true }***.

The most common iterator is the **Array** iterator, which returns each value in the associated array in sequence.

Iterators can express sequences of unlimited size, such as range of integers between 0 and *Infinity*.

Generator

The **Generator** function allows to define an *iterative algorithm* by writing a single function, whose execution is not continuous. Generator functions are written using the ***function**** syntax.

When called, generator functions do not initially execute their code. Instead, they return a special type of iterator, called a **Generator**.

When a value is consumed by calling generator's *next()* method, the generator function executes until it encounters the ***yield*** keyword.

```
function* makeRangeIterator(start = 0,
                            end = Infinity,
                            step =1) {
    let iterationCount = 0;
    for(let i = start, i < end; i++) {
        iterationCount++;
        yield i;
        // yield i returns current
        // value of the sequence
    }
    return iterationCount;
    // iterationCount is returned
    // on completion of execution
}
```

Above *function* can be called as many times as desired, and returns a new *Generator* each time. Each *Generator* may only be iterated once.

```
const it = makeRangeIterator(1, 10, 2);
let result = it.next();

while(!result.done) {
    console.log(result.value);
    // 1  3  5  7  9

    result = it.next();
    // It executes the generator function
    // for next sequence
}

console.log('Sequence of size: ', result.value);
// Sequence of size: 5
```

Note: It is not possible to know reflectively whether a particular object is an iterator. Use **Iterables** to check it.

Iterables

An object is **iterable**, if it defines its iteration behavior, such as what values are looped over in a *for...of* construct. Some built-in types like *Array* and *Map* have a default iteration behavior.

In order to be *iterable*, an object must implement the ***@@iterator*** method.

This means that the object must have a property with a ***Symbol.iterator*** key. It may be possible to iterate over an iterable more than once or only once.

Iterables which can iterate only once (such as *Generators*) return ***this*** from their *@@iterator* method, whereas

iterables which can be iterated many times must return a ***new iterator*** on each invocation of *@@iterator*.

Below code implements an iterable which can iterate only once.

```
function* makeIterator() {
  yield 1;
  yield 2;
}
const it = makeIterator();

for (const itm of it) {
  console.log(itm);
}

//Output:
// 1
// 2
```

Note: If we try to replicate the *for...of* loop, it will not print in the console again, proving that it only iterates once.

```
it[Symbol.iterator]() === it; // true
```

*Because it has the **@@iterator** method **returning itself**, it can iterate **only once**.*
Below code implements an iterable which can iterate many times.

```
function* makeIterator() {
  yield 1;
  yield 2;
}

const it = makeIterator();
it[Symbol.iterator] = function* () {
  yield 1;
  yield 2;
};

for (const itm of it) { console.log(itm); }
// Output:
// 1
// 2

for (const itm of it) { console.log(itm); }
// Output:
// 1
// 2
```

Both for...of loop will print numbers on the console, proving that it can iterate many times.

The **next()** method also accepts a parameter value, which can be used to modify the internal state of the generator. A value passed to *next()* will be received by **yield**.

> **Note**: A value passed to the **first invocation** of *next()* is always **ignored**.

Let's create a fibonacci series generator using the next() method.

```
function* fibonacci() {
  let current = 0;
  let next = 1;
  while (true) {
    const reset = yield current;
    // Value passed to next() will be
    // read here by yield current.

    [current, next] = [next, next + current];
    if (reset) {
      current = 0;
      next = 1;
    }
  }
}
```

Below code uses the defined generator "fibonacci()".

```
const sequence = fibonacci();

sequence.next();
console.log(sequence.next().value); // 1

sequence.next();
console.log(sequence.next().value); // 2

sequence.next();
console.log(sequence.next().value); // 5

sequence.next(true); // Resetting here
console.log(sequence.next().value); // 1
```

Generators have a ***return(value)*** method that returns the given value and finishes the generator itself, and allows the

generator to perform any cleanup tasks when combined with *try...finally* block.

If the yield expression is wrapped in a *try...finally* block, the control flow doesn't exit the function body, but proceeds to the finally block instead.

```
function* generatorFunc() {
  yield 1;
  yield 2;
}

const g = generatorFunc();
g.next(); // { value: 1, dne: false }

g.return('return value');
// { value: 'return value', done: true }
```

Finishes the generator and returns the value passed to return() method.

```
g.next(); // { value: undefined, done: true }
```

*The "value" is **undefined**, because the generator was finished by the **g.return("return value")** statement.*

Summary

- **Iterators** and **Generators** bring the concept of *iteration* and provide a mechanism for customizing behavior of *for...of* loop.
- The **iterator** object implements **iterator protocols** by having ***next()*** methods. Returns an *object* with ***value*** and ***done*** attributes.
- The **Generator** function defines an *iterative algorithm*, whose execution is not continuous. Generator functions use ***function**** syntax.
- The *generator function* executes until it encounters the ***yield*** keyword.
- An object is **iterable**, if it defines *iteration behavior* by implementing the ***@@iterator*** method.
- Iterables which **execute once**, return ***this*** from ***@@iterator***, while one which iterates **multiple times** return **new iterator** from ***@@iterator*** method.
- The ***next()*** method's *optional parameter* is used to modify internal state.
- The ***return()*** method is used to return a *value* and **terminate** the generator.

Glossary

Term	Definition
Coercion	Implicit/Automatic conversion of values from one type to another.
DOM	Document Object Model is a programming interface for the web. Represents documents as nodes and objects.
Unicode	Standard character set of world's different languages, writing systems and symbols.
ASI	Automatic Semicolon Insertion.

References

https://ui-geeks.in an online UI learning platform.

https://developer.mozilla.org an open-source collaborative project documenting Web platform technologies.

Where to go from here

Mastering the topics explained in this book is a great step towards your goal of deep diving JavaScript.

I would suggest continuing your journey with the below action items.

Learn more about **DOM**, **Forms**, **Event Handling**.

Learn how to apply styles to React Components using **CSS/SCSS/LESS**, **Styled Components**, **CSS in JS**.

Start building simple React applications like **Counter** App, **Calculator**, **Tic-Tac-Toe game**, **To-Do** App, etc.

Learn how to test JavaScript applications using **Jest**, **Enzyme**.

Learn **TypeScript**, It is JavaScript with syntax for types.

Liked the Content

If you liked the content of the book and want to support writing more useful content. Your smallest support would keep me motivated to keep writing.

Support Us

Thank You

A book is only the beginning for a writer. A reader completes it, and you are an example of this. Thank you for taking the time and reading the book.

Would appreciate it if you can recommend this to your near and dear ones.

Sunil Kumar

Cell: +91 9818234200
Mail: skumar.mca2010@gmail.com
LinkedIn: sunil-kumar-83146843