More Modern JS Concepts | Cheat Sheet

1. Spread Operator

The Spread Operator is used to unpack an iterable (e.g. an array, object, etc.) into individual elements.

1.1 Spread Operator with Arrays

Example:

```
let arr1 = [2, 3];

let arr2 = [1, ...arr1, 4];

console.log(arr2);  // [1, 2, 3, 4]
```

1.1.1 Creating a Copy

Example:

```
let arr1 = [2, 3];

let arr2 = [...arr1];

console.log(arr2);  // [2, 3]
```

1.1.2 Concatenation

Example:

```
let arr1 = [2, 3];

let arr2 = [4, 5];

let arr3 = [...arr1, ...arr2];

console.log(arr3);  // [2, 3, 4, 5]
```

1.2 Spread Operator with Objects

Example:

```
let person = { name: "Rahul", age: 27 };

let personDetails = { ...person, city: "Hyderabad" };

console.log(personDetails);  // Object {name: "Rahul", age: 27, city: "Hyderabad"}
```

1.2.1 Creating a Copy

Example:

```
let person = { name: "Rahul", age: 27 };

let personDetails = { ...person };

console.log(personDetails);  // Object {name: "Rahul", age: 27}
```

1.2.2 Concatenation

Example:

```
let person = { name: "Rahul", age: 27 };

let address = { city: "Hyderabad", pincode: 500001 };

let personDetails = { ...person, ...address };

console.log(personDetails);  // Object {name: "Rahul", age: 27, city: "Hyderabad", pincode: 500001}
```

1.3 Spread Operator with Function Calls

The Spread Operator syntax can be used to pass an array of arguments to the function. Extra values will be ignored if we pass more arguments than the function parameters.

Example:

```
function add(a, b, c) {

  return a + b + c;

}

let numbers = [1, 2, 3, 4, 5];

console.log(add(...numbers));  // 6
```

2. Rest Parameter

With Rest Parameter, we can pack multiple values into an array.

Example:

```
function numbers(...args) {

  console.log(args);  // [1, 2, 3]

}

numbers(1, 2, 3);
```

<u>2.1 Destructuring arrays and objects with Rest Parameter Syntax</u>

<u>2.1.1 Arrays</u>

```
let [a, b, ...rest] = [1, 2, 3, 4, 5];

console.log(a);  // 1

console.log(b);  // 2

console.log(rest);  // [3, 4, 5]
```

<u>2.1.2 Objects</u>

<u>Example:</u>

```
let { firstName, ...rest } = {

  firstName: "Rahul",

  lastName: "Attuluri",

  age: 27

};

console.log(firstName);  // Rahul

console.log(rest);  // Object {lastName: "Attuluri", age: 27}
```

<u>Note:</u>

The Rest parameter should be the last parameter.

<u>Example-1:</u>

```
function numbers(a, b, ...rest) {

  console.log(a);  // 1

  console.log(b);  // 2

  console.log(rest);  // [3, 4, 5]

}

numbers(1, 2, 3, 4, 5);
```

<u>Example-2:</u>

```
function numbers(a, ...rest, b) {

  console.log(a);
  console.log(rest);

  console.log(b);

}

numbers(1, 2, 3, 4, 5);  // Uncaught SyntaxError: Rest parameter must be last formal parameter
```

3. Functions

3.1 Default Parameters

The Default Parameters allow us to give default values to function parameters.

Example:

```
function numbers(a = 2, b = 5) {

  console.log(a);  // 3

  console.log(b);  // 5

}

numbers(3);
```

4. Template Literals (Template Strings)

The Template Literals are enclosed by the backticks.

They are used to:

1. Embed variables or expressions in the strings

2. Write multiline strings

We can include the variables or expressions using a dollar sign with curly braces **${ }.**

Example:

```
let firstName = "Rahul";

console.log(`Hello ${firstName}!`);  // Hello Rahul!
```

More Modern JS Concepts Part 2 | Cheat Sheet

1. Operators

1.1 Ternary Operator

A Ternary Operator can be used to replace **if...else** statements in some situations.

Syntax: condition ? expressionIfTrue : expressionIfFalse

Example:

```
let speed = 70;

let message = speed >= 100 ? "Too Fast" : "OK";

console.log(message);  // OK
```

2. Conditional Statements

2.1 Switch Statement

A Switch statement is a conditional statement like **if...else** statement used in decision making.

Syntax:

```
switch (expression) {
  case value1:
    /*Statements executed when the
    result of expression matches value1*/
    break;
  case value2:
    /*Statements executed when the
    result of expression matches value2*/
    break;
  ...
  case valueN:
    /*Statements executed when the
    result of expression matches valueN*/
    break;
  default:
    /*Statements executed when none of
    the values match the value of the expression*/
    break;
}
```

Example:

```
let day = 1;

switch (day) {

 case 0:

  console.log("Sunday");

  break;

 case 1:

  console.log("Monday");  // Monday

  break;

 case 2:

  console.log("Tuesday");

  break;

 case 3:

  console.log("Wednesday");

  break;

 case 4:

  console.log("Thursday");

  break;

 case 5:

  console.log("Friday");

  break;

 case 6:

  console.log("Saturday");

  break;

 default:

  console.log("Invalid");

  break;
}
```

2.1.1 What happens if we forgot a break?

If there is no break statement, then the execution continues with the next case until the break statement is met.

3. Defining Functions

There are multiple ways to define a function.

- Function Declaration
- Function Expression
- Arrow Functions
- Function Constructor, etc.

3.1 Arrow Functions

An Arrow function is a simple and concise syntax for defining functions.

It is an alternative to a function expression.

Syntax:

```
let sum = (param1, param2, …) => {

  // statement(s)

};

sum();
```

Example:

```
let sum = (a, b) => {

  let result = a + b;

  return result;

};

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

3.1.1 Simple Expressions

In arrow functions, the return statement and curly braces are not required for simple expressions.

Example:

```
let sum = (a, b) => a + b;
```

```
console.log(sum(4, 3));  // 7
```

3.1.2 One parameter

If there is only one parameter, then parentheses are not required.

Example:

```
let greet = name => `Hi ${name}!`;

console.log(greet("Rahul"));  // Hi Rahul!
```

3.1.3 No parameters

If there are no parameters, parentheses will be empty, but they should be present.

Example:

```
let sayHi = () => "Hello!";

console.log(sayHi());  // Hello!
```

3.1.4 Returning Objects

Example:

```
let createUser = name => {

  return {

    firstName: name

  };

};

console.log(createUser("Rahul"));  // Object {firstName: "Rahul"}
```

Simple Expression:

```
let createUser = name => { firstName: "Rahul" };

console.log(createUser());  // undefined
```

JavaScript considers the two curly braces as a code block, but not as an object syntax.

So, wrap the object with parentheses to distinguish with a code block.

```
let createUser = name => ({ firstName: "Rahul" });

console.log(createUser());  // Object {firstName: "Rahul"}
```

More JS Concepts | Cheat Sheet

1. Scoping

The Scope is the region of the code where a certain variable can be accessed.

In JavaScript there are two types of scope:

- Block scope
- Global scope

1.1 Block Scope

If a variable is declared with const or let within a curly brace ({}), then it is said to be defined in the **Block Scope**.

- if..else
- function (){}
- switch
- for..of, etc.

Example :

```
let age = 27;

if (age > 18) {

  let x = 0;

  console.log(x); // 0

}

console.log(x); //  ReferenceError{"x is not defined"}
```

1.2 Global Scope

- If a variable is declared outside all functions and curly braces ({}), then it is said to be defined in the **Global Scope**.
- When a variable declared with **let or const** is accessed, Javascript searches for the variable in the block scopes first followed by global scopes.

Example:

```
const x = 30;

function myFunction() {
  if (x > 18) {
    console.log(x); // 30
  }
}
myFunction();
```

2. Hoisting

2.1 Function Declarations

Hoisting is a JavaScript mechanism where **function declarations** are moved to the top of their scope before code execution.

Example:

```
let x = 15;
let y = 10;
let result = add(x, y);
console.log(result); // 25
function add(a, b) {
  return a + b;
}
```

2.2 Function Expressions

Function expressions in JavaScript are not hoisted.

Example:

```
myFunction();
let myFunction = function () {
  let x = 5;
  console.log(x); // ReferenceError{"Cannot access 'myFunction' before initialization"}
};
```

2.3 Arrow Functions

Arrow Functions in JavaScript are not hoisted.

Example:

```
myFunction();

let myFunction = () => {

  let x = 5;

  console.log(x); // ReferenceError{"Cannot access 'myFunction' before initialization"}

};
```

3. More Array Methods

The **map(), forEach(), filter()** and **reverse()** are some of the most commonly used array methods in JavaScript.

3.1 Map()

The **map()** method creates a new array with the results of calling a function for every array element.

The **map()** method calls the provided function once for each element in an array, in order.

Syntax : array.map(callback(currentValue, index, arr))

Here the **callback** is a function that is called for every element of array.

currentValue is the value of the current element and **index** is the array index of the current element. Here index and **arr** are optional arguments.

Example:

```
const numbers = [1, 2, 3, 4];

const result = numbers.map((number) => number * number);

console.log(result); // [1, 4, 9, 16]
```

3.2 forEach()

The **forEach()** method executes a provided function once for each array element. It always returns undefined.

Syntax : array.forEach(callback(currentValue, index, arr))

Here **index** and **arr** are optional arguments.

Example:

```
let fruits = ["apple", "orange", "cherry"];
```

fruits.forEach((fruit) => console.log(fruit));

3.3 filter()

The **filter()** method creates a new array filled with all elements that pass the test (provided as a function).

A new array with the elements that pass the test will be returned. If no elements pass the test, an empty array will be returned.

Syntax : array.filter(function(currentValue, index, arr))

Here **index** and **arr** are optional arguments.

Example:

const numbers = [1, -2, 3, -4];

const positiveNumbers = numbers.filter((number) => number > 0);

console.log(positiveNumbers); // **[1, 3]**

3.4 reduce()

The **reduce()** method executes a reducer function (that you provide) on each element of the array, resulting in single output value.

Syntax : array.reduce(function(accumulator, currentValue, index, arr), initialValue)

Here **accumulator** is the **initialValue** or the previously returned value of the function and **currentValue** is the value of the current element, **index** and **arr** are optional arguments.

Example:

const array1 = [1, 2, 3, 4];

const reducer = (accumulator, currentValue) => accumulator + currentValue;

console.log(array1.reduce(reducer)); // **10**

3.5 every()

The **every()** method tests whether all elements in the array pass the test implemented by the provided function. It returns a Boolean value.

Syntax : array.every(function(currentValue, index, arr))

Here **index** and **arr** are optional arguments.

Example:

let array1 = [32, 33, 16, 20];

```
const result = array1.every((array1) => array1 < 40);

console.log(result); // true
```

3.6 some()

The **some()** method tests whether at least one element in the array passes the test implemented by the provided function. It returns a Boolean value.

Syntax : array.some(function(currentValue, index, arr))

Here **index** and **arr** are optional arguments.

Example:

```
const myAwesomeArray = ["a", "b", "c", "d", "e"];

const result = myAwesomeArray.some((alphabet) => alphabet === "d");

console.log(result); // true
```

3.7 reverse()

The **reverse()** method reverses the order of the elements in an array.The first array element becomes the last, and the last array element becomes the first.

Syntax : array.reverse()

Example:

```
const myArray = ["iBHubs", "CyberEye", "ProYuga"];

const reversedArray = myArray.reverse();

console.log(reversedArray); // ["ProYuga", "CyberEye", "iBHubs"]
```

3.8 flat()

The **flat()** method creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.

Syntax : let newArray = arr.flat([depth]);

Example:

```
const arr1 = [0, 1, 2, [3, 4]];

const arr2 = [0, 1, 2, [[[3, 4]]]];

console.log(arr1.flat()); // [ 0,1,2,3,4 ]

console.log(arr2.flat(2)); // [0, 1, 2, [3, 4]]
```

4. Mutable & Immutable methods

Mutable methods will change the original array and Immutable methods won't change the original array.

Mutable methods	Immutable methods
shift()	map()
unshift()	filter()
push()	reduce()
pop()	forEach()
sort()	slice()
reverse()	join()
splice(), etc.	some(), etc.

Factory and Constructor Functions | Cheat Sheet

1. Factory Function

A Factory function is any function that returns a new object for every function call.

The Function name should follow the **camelCase** naming convention.

Syntax:

```
function functionName(parameter1, parameter2, ...) {
  return {
    property1: parameter1,
    property2: parameter2,
    ...
    ...
  }
}
let myObject = functionName(arg1, arg2, ..)
```

Example:

```
function createCar(color, brand) {
  return {
    color: color,
    brand: brand,
    start: function() {
      console.log("started");
    }
  };
}
let car1 = createCar("blue", "Audi");
let car2 = createCar("red", "Tata");
let car3 = createCar("green", "BMW");
```

```
console.log(car1);  // Object { color: "blue", brand: "Audi", start: ƒ() }

console.log(car2);  // Object { color: "red", brand: "Tata", start: ƒ() }

console.log(car3);  // Object { color: "green", brand: "BMW", start: ƒ() }
```

1.1 <u>Shorthand Notations:</u>

<u>Eaxmple:</u>

```
function createCar(color, brand) {
  return {
    color,
    brand,
    start() {
      console.log("started");
    }
  };
}


let car1 = createCar("blue", "Audi");

let car2 = createCar("red", "Tata");

let car3 = createCar("green", "BMW");

console.log(car1);  // Object { color: "blue", brand: "Audi", start: ƒ() }

console.log(car2);  // Object { color: "red", brand: "Tata", start: ƒ() }

console.log(car3);  // Object { color: "green", brand: "BMW", start: ƒ() }
```

<u>2. Constructor Function</u>

A regular function that returns a new object on calling with the new operator. The created new object is called an Instance.

The Function name should follow the **PascalCase** naming convention.

<u>Syntax:</u>

```
function FunctionName(parameter1, parameter2, ...) {

  this.property1 = parameter1;

  this.property2 = parameter2;

  ...

  ...

}

let myObject = new FunctionName(arg1, arg2,…)
```

2.1 The new Operator

When a function is called with the **new** operator, it does the following steps:

- Creates an empty object and assigns it to **this**
- Return **this**

Example:

```
function Car(color, brand) {

  this.color = color;

  this.brand = brand;

  this.start = function() {

    console.log("started");

  };

}

let car1 = new Car("blue", "Audi");

console.log(car1);  // Car { }
```

Here,

car1 is instance

car1.start() is instance method

car1.color, **car1.brand** are instance properties

2.2 Factory vs Constructor Functions

Factory Functions	Constructor Functions
Follows **camelCase** notation	Follows **PascalCase** notation
Doesn't need `new` operator for function calling	Needs `new` operator for function calling
Explicitly need to return the object	Created object returns implicitly

3. JS Functions

Similar to Objects, Functions also have properties and methods.

3.1 Default Properties

- name
- length
- constructor
- prototype, etc.

3.2 Default Methods

- apply()
- bind()
- call()
- toString(), etc.

3.3 Function Properties

3.3.1 The name Property

This property returns the name of the function.

Example:

```
function Car(color, brand) {
  this.color = color;
  this.brand = brand;
  this.start = function() {
    console.log("started");
  };
}
console.log(Car.name);  // Car
```

3.3.2 The length Property

This property returns the number of parameters passed to the function.

Example:

```
function Car(color, brand) {
  this.color = color;
  this.brand = brand;
  this.start = function() {
    console.log("started");
  };
}
console.log(Car.length);  // 2
```

3.3.3 The typeof function

Example:

```
function Car(color, brand) {
  this.color = color;
  this.brand = brand;
  this.start = function() {
    console.log("started");
  };
}
console.log(typeof(Car));  // function
```

4. The Constructor Property

Every object in JavaScript has a constructor property.

The constructor property refers to the constructor function that is used to create the object.

Example:

```
function Car(color, brand) {
  this.color = color;
  this.brand = brand;
  this.start = function() {
    console.log("started");
  };
}
let car1 = new Car("blue", "Audi");
console.log(car1.constructor);  // f Car(color, brand) {}
```

5. Built-in Constructor Function

These are the Constructor functions provided by JavaScript.

- **function Date()**
- function Error()
- function Promise()
- function Object()
- function String()
- function Number(), etc.

In JavaScript, date and time are represented by the Date object. The Date object provides the date and time information and also provides various methods.

5.1 Creating Date Objects

There are four ways to create a date object.

- new Date()
- new Date(milliseconds)
- new Date(datestring)
- new Date(year, month, day, hours, minutes, seconds, milliseconds)

5.1.1 new Date()

You can create a date object without passing any arguments to the **new Date()** constructor function.

For example,

let now = new Date();

console.log(now); // **Tue Feb 02 2021 19:10:29 GMT+0530 (India Standard Time) { }**

console.log(typeof(now)); // **object**

Here, **new Date()** creates a new date object with the current date and local time.

Note:

1. Coordinated Universal Time (UTC) - It is the global standard time defined by the World Time Standard. (This time is historically known as Greenwich Mean Time, as UTC lies along the meridian that includes London and nearby Greenwich in the United Kingdom.)

2. Local Time - The user's device provides the local time.

5.1.2 new Date(milliseconds)

The Date object contains a number that represents milliseconds since **1 January 1970 UTC**.

The **new Date(milliseconds)** creates a new date object by adding the milliseconds to zero time.

For example,

let time1 = new Date(0);

console.log(time1); // **Thu Jan 01 1970 05:30:00 GMT+0530 (India Standard Time) { }**

// 100000000000 milliseconds from 1 Jan 1970 UTC

let time2 = new Date(100000000000);

console.log(time2); // **Sat Mar 03 1973 15:16:40 GMT+0530 (India Standard Time) { }**

Note: 1000 milliseconds is equal to 1 second.

5.1.3 new Date(date string)

The **new Date(date string)** creates a new date object from a date string.

Syntax: **new Date(datestring);**

Example:

let date = new Date("2021-01-28");

console.log(date); // **Thu Jan 28 2021 05:30:00 GMT+0530 (India Standard Time) { }**

You can also pass only the year and month or only the year. For example,

```
let date = new Date("2020-08");

console.log(date);  // Sat Aug 01 2020 05:30:00 GMT+0530 (India Standard Time) { }


let date1 = new Date("2020");

console.log(date1);  // Wed Jan 01 2020 05:30:00 GMT+0530 (India Standard Time) { }
```

Short date format

```
// short date format "MM/DD/YYYY"

let date = new Date("03/25/2015");

console.log(date);  // Wed Mar 25 2015 00:00:00 GMT+0530 (India Standard Time) { }
```

Long date format

```
// long date format "MMM DD YYYY"

let date1 = new Date("Jul 1 2021");

console.log(date1);  // Thu Jul 01 2021 00:00:00 GMT+0530 (India Standard Time) { }
```

Month and Day can be in any order

```
let date2 = new Date("1 Jul 2021");

console.log(date2);  // Thu Jul 01 2021 00:00:00 GMT+0530 (India Standard Time) { }
```

The month can be full or abbreviated. Also, month names are case insensitive.

```
let date3 = new Date("July 1 2021");

console.log(date3);  // Thu Jul 01 2021 00:00:00 GMT+0530 (India Standard Time) { }


// commas are ignored

let date4 = new Date("JULY, 1, 2021");

console.log(date4);  // Thu Jul 01 2021 00:00:00 GMT+0530 (India Standard Time) { }
```

5.1.4 new Date(year, month, day, hours, minutes, seconds, milliseconds)

It creates a new date object by passing a specific date and time.

For example,

let time1 = new Date(2021, 1, 20, 4, 12, 11, 0);

console.log(time1); // **Sat Feb 20 2021 04:12:11 GMT+0530 (India Standard Time) { }**

Similarly, if two arguments are passed, it represents year and month.

For example,

let time1 = new Date(2020, 1);

console.log(time1); // Sat Feb 20 2021 04:00:00 GMT+0530 (India Standard Time) { }

Warning

If you pass only one argument, it is treated as milliseconds. Hence, you have to pass two arguments to use this date format.

5.2 AutoCorrection in Date Object

When you assign out of range values in the Date object, it auto-corrects itself.

For example,

let date = new Date(2008, 0, 33);

// Jan does not have 33 days

console.log(date); // **Sat Feb 02 2008 00:00:00 GMT+0530 (India Standard Time) { }**

33 days are auto corrected to 31 (jan) + 2 days in feb.

5.3 Instance Methods

There are methods to access and set values like a year, month, etc. in the Date Object.

Method	Description
now()	Returns the numeric value corresponding to the current time (the number of milliseconds passed since January 1, 1970, 00:00:00 UTC)
getFullYear()	Gets the year according to local time
getMonth()	Gets the month, from 0 to 11 according to local time
getDate()	Gets the day of the month (1–31) according to local time
getDay()	Gets the day of the week (0-6) according to local time
getHours()	Gets the hour from 0 to 23 according to local time

Method	Description
getMinutes	Gets the minute from 0 to 59 according to local time
getUTCDate()	Gets the day of the month (1–31) according to universal time
setFullYear()	Sets the full year according to local time
setMonth()	Sets the month according to local time
setDate()	Sets the day of the month according to local time
setUTCDate()	Sets the day of the month according to universal time

Example:

```
let date1 = new Date(1947, 7, 15, 1, 3, 15, 0);

console.log(date1.getFullYear());  // 1947

console.log(date1.getMonth());  // 7
```

5.3.1 Setting Date Values

```
let date1 = new Date(1947, 7, 15);

date1.setYear(2021);

date1.setDate(1);

console.log(date1);  // Sun Aug 01 2021 00:00:00 GMT+0530 (India Standard Time) { }
```

More Modern JS Concepts | Part 3 | Cheat Sheet

1. this

The this is determined in three ways.

- In Object methods, it refers to the object that is executing the current function.
- In Regular functions, it refers to the window object.
- In Arrow functions, it refers to the context in which the code is defined.

1.1 this in Object Methods

Example:

```
let car = {
  color: "blue",

  brand: "Audi",

  start: function() {

    console.log(this);  // Object { color: "blue", brand: "Audi", start: ƒ() }

  }

};

car.start();
```

In the above example, this refers to the car object as it's executing the method start.

1.2 <u>this in Regular Functions</u>

<u>Example:</u>

```
function start() {

  console.log(this);  // Window { }

}

start();
```

In the above example, this refers to the window object.

<u>1.3 this in Arrow Functions</u>

In Arrow functions, this depends on two aspects:

- When the code is defined
- Context

Arrow function inherits this from the context in which the code is defined.

<u>1.3.1 Object Methods</u>

<u>Example:</u>

```
let car = {

  color: "blue",

  brand: "Audi",

  start: () => {

console.log(this);  // Window { }

  }
};

car.start();
```

Arrow Functions with Callbacks

```
let car = {
  color: "blue",
  brand: "Audi",
  start: function() {
    setTimeout(() => {
      console.log(this);  // Object { color: "blue", brand: "Audi", start: ƒ() }
    }, 1000);
  }
};

car.start();
```

1.4 this in Constructor Functions

In Constructor Function, **this** refers to the instance object.

Example:

```
function Car(color, brand) {
  this.color = color;
  this.brand = brand;
  this.start = function() {
    console.log(this);  // Car { }
  };
}

let car1 = new Car("blue", "Audi");

car1.start();
```

In the above example, this refers to the object car1.

1.4.1 Arrow Functions

Example:

```
function Car(color, brand) {
  this.color = color;
  this.brand = brand;
  this.start = () => {
    console.log(this);  // Car { }
  };
}
let car1 = new Car("blue", "Audi");
car1.start();
```

2. Immutable and Mutable Values

2.1 Immutable

All the primitive type values are immutable.

- Number
- String
- Boolean
- Symbol
- Undefined, etc.

Example:

```
let x = 5;
let y = x;
y = 10;
console.log(x);  // 5
console.log(y);  // 10
```

2.2 Mutable

All the Objects are mutable.

- Object

- Array
- Function

Example:

```
let x = { value: 5 };

let y = x;

let z = { value: 20 };

y.value = 10;

console.log(x);  // Object { value: 10 }

console.log(y);  // Object { value: 10 }

y = z;

console.log(x);  // Object { value: 10 }

console.log(y);  // Object { value: 20 }
```

3. Declaring Variables

In JavaScript, a variable can be declared in 3 ways.

- Using let
- Using const
- Using var

3.1 let

While declaring variables using let,

- Initialization is not mandatory
- Variables can be reassigned

Example:

```
let x;

x = 10;

console.log(x);  // 10

x = 15;

console.log(x);  // 15
```

3.2 const

While declaring variables using const,

- Initialization is mandatory
- Once a value is initialized, then it can't be reassigned

Without Initialization:

```
 const x;

  x = 7;  // SyntaxError {"Const declarations require an initialization value (1:21)"}
```

Reassignment:

```
const x = 7;

x = 9;  // TypeError {"Assignment to constant variable."}
```

3.2.1 Mutating Object properties

Example:

```
const car = {

 color : "blue",

 brand : "Audi"

};

car.color = "red";

console.log(car.color);  // red
```

But objects can't be reassigned

```
const car = {

 color : "blue",

 brand : "Audi"

};

car.color = "red";

car = {};  // TypeError {"Assignment to constant variable."}
```

Prototypal Inheritance | Cheat Sheet

<u>1. Built-in Constructor Functions</u>

These are the built-in constructor functions provided by JavaScript.

- **function Array()**
- **function Function()**
- function Promise()
- function Object()
- function String()
- function Number(), etc.

<u>2. Built-in</u>

<u>Array</u>

Constructor Function

2.1 Default Properties and Methods

<u>Properties:</u>

- constructor
- length
- prototype, etc.

<u>Methods:</u>

- push()
- pop()
- splice()
- shift(), etc.

<u>2.2 Creating an Array with the new Operator (Older way of writing)</u>

<u>Syntax:</u>

let myArray = new Array(item1, item2, ...);

<u>Example:</u>

let myArray = new Array("a", 2, true);

myArray.push("pen");

console.log(myArray); // **Array (4)["a", 2, true, "pen"]**

console.log(myArray.length); // **4**

3. Prototype Property

The Prototype property will be **shared** across all the **instances** of their constructor function.

3.1 Accessing the Prototype of a Constructor Function

Example: console.log(Array.prototype);

3.2 Accessing the shared Prototype of an Instance

Example: let myArray = new Array("a", 2, true);

console.log(Object.getPrototypeOf(myArray));

3.3 Prototypal Inheritance

On calling the **new()** operator, all the properties and methods defined on the prototype will become accessible to the instance objects. This process is called Prototypal Inheritance.

4. Built-in Function Constructor Function

4.1 Default Properties and Methods

Properties:

- name
- length
- constructor
- prototype, etc.

Methods:

- apply()
- bind()
- call()
- toString(), etc.

4.2 Creating a Function with the new Operator (Older way of writing)

Syntax: let myFunction = new Function("param1, param2, ...", function body);

Example:

```
let Car = new Function("color, brand",
  `this.color = color;
   this.brand = brand;
   this.start = function() {
     console.log("started");
  };`);
console.log(Function.prototype);
```

5. Instance Specific and Prototype Properties

5.1 Prototype Properties/ Methods

The Prototype Properties/ Methods are the properties or methods common across the instance objects.

Examples:

- calculateAge
- displayGreetings
- displayProfileDetails
- calculateIncome

5.1.1 Adding a Method to the prototype

Example:

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}
Person.prototype.displayFullName = function() {
  return this.firstName + " " + this.lastName;
};
let person1 = new Person("Virat", "Kohli");
let person2 = new Person("Sachin", "Tendulkar");
```

```
console.log(Object.getPrototypeOf(person1) === Object.getPrototypeOf(person2));
```

5.2 Instance Specific Properties/ Methods

The Instance Specific Properties/ Methods are the properties or methods specific to the instance object.

Examples:

- gender
- yearOfBirth
- friendsList
- name

Example:

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}
Person.prototype.displayFullName = function() {
  return this.firstName + " " + this.lastName;
};
let person1 = new Person("Virat", "Kohli");
console.log(Object.getOwnPropertyNames(person1));
```

JS Classes | Cheat Sheet

1. Class

The class is a special type of function used for creating multiple objects.

1.1. Constructor Method

The constructor method is a special method of a class for creating and initializing an object of that class.

Syntax:

```
class MyClass {
  constructor(property1, property2) {
    this.property1 = property1;

    this.property2 = property2;

  }

  method1() { ... }

  method2() { ... }

}
```

1.1.1 Creating a Single Object

Syntax :

```
class MyClass {

  constructor(property1, property2) {

    this.property1 = property1;

    this.property2 = property2;

  }

  method1() { ... }

  method2() { ... }

}

let myObject = new MyClass(property1, property2);
```

Example :

```
class Person {

  constructor(firstName, lastName) {

    this.firstName = firstName;

    this.lastName = lastName;

 }

 displayFullName() {

   return this.firstName + " " + this.lastName;

 }

}
```

```
let person1 = new Person("Virat", "Kohli");

console.log(person1);  // Person {...}
```

1.1.2 Creating Multiple Objects

Example:

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
}
let person1 = new Person("Virat", "Kohli");
let person2 = new Person("Sachin", "Tendulkar");
console.log(person1);  // Person {...}
console.log(person2);  // Person {...}
```

1.2 Prototype property of a Class

Example:

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
  displayFullName() {
   return this.firstName + " " + this.lastName;
  }
}
let person1 = new Person("Virat", "Kohli");
console.log(Person.prototype);  // Person {...}
```

1.3 Prototype of an Instance

The Instance Prototype refers to the prototype object of the constructor function.

Example:

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
  displayFullName() {
   return this.firstName + " " + this.lastName;
  }
}
let person1 = new Person("Virat", "Kohli");
console.log(Object.getPrototypeOf(person1));  // Person {...}
```

Note

The Type of a class is a function

2.Inheritance in JS Classes

The Inheritance is a mechanism by which a class inherits methods and properties from another class.

2.1 Extends

The extends keyword is used to inherit the methods and properties of the superclass.

2.2 Super

Calling super() makes sure that SuperClass constructor() gets called and initializes the instance.

Syntax :

```
class SuperClass {
}
class SubClass extends SuperClass{
  constructor(property1, property2){
    super(property1);
    this.property2 = property2;
  }
  method1() {  }
}
let myObject = new SubClass(property1, property2);
```

Here, SubClass inherits methods and properties from a SuperClass.

2.3 Method Overriding

Here the constructor method is overridden. If we write the SuperClass methods in SubClass, it is called method overriding.

Syntax :

```
class SuperClass {
}
class SubClass extends SuperClass{
  constructor(property1, property2){
    super(property1);
    this.property2 = property2;
  }
}
let myObject = new SubClass(property1, property2);
```

Example:

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  eat() {
    return `${this.name} is eating`;
  }
  makeSound() {
    return `${this.name} is shouting`;
  }
}
class Dog extends Animal {
  constructor(name, breed) {
    super(name);
    this.breed = breed;
  }
  sniff() {
    return `${this.name} is sniffing`;
  }
}
class Cat extends Animal {
  constructor(name, breed) {
    super(name);
    this.breed = breed;
  }
  knead() {
    return `${this.name} is kneading`;
  }
}
let animal1 = new Animal("gorilla");
let someDog = new Dog("someDog", "German Shepherd");
let persianCat = new Cat("someCat", "Persian Cat");

console.log(animal1);  // Animal {...}
```

console.log(animal1.eat()); // gorilla is eating

console.log(someDog); // **Dog {...}**

console.log(someDog.eat()); // **someDog is eating**

console.log(someDog.sniff()); // **someDog is sniffing**

console.log(persianCat); // Cat {...}

console.log(persianCat.knead()); **// someCat is kneading**

console.log(persianCat.eat()); // **someCat is eating**

console.log(persianCat.makeSound()); // **someCat is shouting**

3.this in classes

3.1 Super Class

In class, this refers to the instance object.

Example:

```
class Animal {
  constructor(name) {
    this.name = name;
 }
 eat() {
   return this;
 }
 makeSound() {
   return `${this.name} is shouting!`;
 }
}
let animal1 = new Animal("dog");
```

console.log(animal1.eat()); // **Animal {...} Here this refers to the animal1.**

<u>3.2 Sub Class</u>

<u>Example:</u>

```
class Animal {
    constructor(name) {
    this.name = name;
  }
}
class Dog extends Animal {
  constructor(name, breed) {
    super(name);
    this.breed = breed;
  }
  sniff() {
    return this;
  }
}
let dog = new Dog("dog", "german Shepherd");
console.log(dog.sniff());  // Dog {...}
```

JS Promises | Cheat Sheet

1. Synchronous Execution

Example :

```
alert("First Line");

alert("Second Line");

alert("Third Line");
```

The code executes line by line. This behavior is called synchronous behavior, in JS alert works synchronously.

2. Asynchronous Execution

Example 1:

```
const url = "https://apis.ccbp.in/jokes/random";

fetch(url)

  .then((response) => {

    return response.json();

  })

  .then((jsonData) => {

    //statement-1

    console.log(jsonData); // Object{ value:"The computer tired when it got home because it had a hard drive" }

  });

//statement-2

console.log("fetching done"); // fetching done
```

In the above example, the second statement won't wait until the first statement execution. In JS, fetch() works asynchronously.

3. JS Promises

1. Promise is a way to handle **Asynchronous** operations.
2. A promise is an object that represents a result of operation that will be returned at some point in the future.

Example :

```
const url = "https://apis.ccbp.in/jokes/random";

let responseObject = fetch(url);

console.log(responseObject); // **Promise{ [[PromiseStatus]]:pending, [[PromiseValue]]:undefined }**

console.log("fetching done"); // **fetching done**
```

Note

A promise will be in any one of the three states:

1. **Pending :** Neither fulfilled nor rejected
2. **Fulfilled :** Operation completed successfully
3. **Rejected :** Operation failed

3.1 Resolved State

When a Promise object is Resolved, the result is a value.

Example:

```
const url = "https://apis.ccbp.in/jokes/random";

let responsePromise = fetch(url);

responsePromise.then((response) => {

  console.log(response); // Response{ … }

});
```

3.2 Rejected State

Fetching a resource can be failed for various reasons like:

- URL is spelled incorrectly
- Server is taking too long to respond
- Network failure error, etc.

Example:

```
const url = "https://a.ccbp.in/random";

let responsePromise = fetch(url);

responsePromise.then((response) => {

  return response;
});
responsePromise.catch((error) => {
  console.log(error); // TypeError{ "Failed to fetch" }
});
```

3.3 Promise Chaining

Combining multiple .**then()**s or .**catch()**s to a single promise is called promise chaining.

Syntax :

```
const url = "INCORRECT_RESOURCE_URL";
let responsePromise = fetch(url);
responsePromise
  .then((response) => {
    console.log(response);
  })
  .catch((error) => {
    console.log(error);
  });
```

3.3.1 OnSuccess Callback returns Promise

Here, log the response in JSON format.

Example:

```
const url = "RESOURCE_URL";
let responsePromise = fetch(url);
responsePromise.then((response) => {
  console.log(response.json());
});
```

3.3.2 Chaining OnSuccess Callback again

Example:

```
const url = "https://apis.ccbp.in/jokes/random";
```

```
let responsePromise = fetch(url);

responsePromise

  .then((response) => {

    return response.json();

  })

  .then((data) => {

    console.log(data);

  });
```

3.4 Fetch with Error Handling

Check the behavior of code with valid and invalid URLs.

Example:

```
const url = "https://apis.ccbp.in/jokes/random";

let responsePromise = fetch(url);

responsePromise

  .then((response) => {

    return response.json();

  })

  .then((data) => {

    console.log(data); // Object { value: "They call it the PS4 because there are only 4 games worth playing!"

  })

  .catch((error) => {

    console.log(error);

  });
```

JS Promises | Part 2 | Cheat Sheet

1. Asynchronous JS code style

There are two main types of asynchronous code style you'll come across in JavaScript:

Callback based

Example : setTimeout(), setInterval()

Promise based

Example : fetch()

2. Creating your own Promises

Promises are the new style of **async** code that you'll see used in modern JavaScript.

Syntax :

```
const myPromise = new Promise((resolveFunction, rejectFunction) => {
   //Async task
});
```

In the above syntax:

- The Promise constructor takes a function (an executor) that will be executed immediately and passes in two functions: resolve, which must be called when the Promise is resolved (passing a result), and reject when it is rejected (passing an error).
- The *executor* is to be executed by the constructor, during the process of constructing the new Promise object.
- resolveFunction is called on promise fulfilled.
- rejectFunction is called on promise rejection.

Example :

```
const myPromise = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve();
    }, 1000);
  });
};

console.log(myPromise());
```

2.1 Accessing Arguments from Resolve

When **resolve()** is excuted, callback inside **then()** will be executed.

Example :

```
const myPromise = () => {

  return new Promise((resolve, reject) => {

    setTimeout(() => {

      resolve("Promise Resolved");

    }, 1000);

  });

};

myPromise().then((fromResolve) => {

  console.log(fromResolve); // Promise Resolved

});
```

2.2 Accessing Arguments from Reject

When **reject()** is excuted, callback inside **catch()** will be executed.

Example :

```
const myPromise = () => {

  return new Promise((resolve, reject) => {

    setTimeout(() => {

      reject("Promise Rejected");

    }, 2000);

  });

};

myPromise()

  .then((fromResolve) => {
```

```
  console.log(fromResolve);
 })
 .catch((fromReject) => {
  console.log(fromReject); // Promise Rejected
 });
```

3. Async/Await

1. The Async/Await is a **modern way** to consume promises.
2. The **Await** ensures processing completes before the next statement executes.

Syntax :

```
const myPromise = async () => {
 let promiseObj1 = fetch(url1);
 let response1 = await promiseObj1;
 let promiseObj2 = fetch(url2);
 let response2 = await promiseObj2;
};
myPromise();
```

Note

1. Use **async** keyword *before the function* only if it is performing async operations.
2. Should use **await** inside an **async** function only.

3.1 Fetch with Async and Await

Example :

```
const url = "https://apis.ccbp.in/jokes/random";
const doNetworkCall = async () => {
 const response = await fetch(url);
 const jsonData = await response.json();
 console.log(jsonData);
};

doNetworkCall();
```

3.2 Error Handling with Async and Await

Example :

```
const url = "https://a.ccbp.in/jokes/random";

const doNetworkCall = async () => {

  try {

    const response = await fetch(url);

    const jsonData = await response.json();

    console.log(jsonData);

  } catch (error) {

    console.log(error);

  }

};

doNetworkCall();
```

3.3 Async Function always returns Promise

Example :

```
const url = "https://apis.ccbp.in/jokes/random";

const doNetworkCall = async () => {

  const response = await fetch(url);

  const jsonData = await response.json();

  console.log(jsonData);

};

const asyncPromise = doNetworkCall();

console.log(asyncPromise);
```

4. String Manipulations

There are methods and properties available to all strings in JavaScript.

String Methods	Functionality
toUpperCase(), toLowerCase()	Converts from one case to another
includes(), startsWith(), endsWith()	Checks a part of the string
split()	Splits a string
toString()	Converts number to a string
trim(), replace()	Updates a string
concat(), slice(), substring()	Combines & slices strings
indexOf()	Finds an index

4.1 trim()

The **trim( )** method removes whitespace from both ends of a string.

Syntax : string.trim()

Example:

```
const greeting = "   Hello world!  ";

console.log(greeting);

console.log(greeting.trim());
```

4.2 slice()

The **slice()** method extracts a section of a string and returns it as a new string, without modifying the original string.

Syntax : string.slice(start, end)

Example:

```
const text = "The quick brown fox";

console.log(text.slice(0, 3)); // The

console.log(text.slice(2, 3)); // e
```

4.3 toUpperCase()

The toUpperCase() method converts a string to upper case letters.

Syntax : string.toUpperCase()

Example:

const text = "The quick brown fox";

console.log(text.toUpperCase()); // **THE QUICK BROWN FOX**

4.4 toLowerCase()

The toLowerCase() method converts a string to lower case letters.

Syntax : string.toLowerCase()

Example:

const text = "Learn JavaScript";

console.log(text.toLowerCase()); // **learn javascript**

4.5 split()

The **split()** method is used to split a string into an array of substrings and returns the new array.

Syntax : string.split(separator, limit)

Example:

const str = "He-is-a-good-boy";

const words = str.split("-");

console.log(words); // **["He", "is", "a", "good", "boy"]**

4.6 replace()

The replace() method searches a string for a specified value, or a regular expression, and returns a new string where the specified values are replaced.

Syntax : string.replace(specifiedvalue, newvalue)

Example:

const str = "Welcome to Hyderabad";

const words = str.replace("Hyderabad", "Guntur");

console.log(words); // **Welcome to Guntur**

4.7 includes()

The **includes()** method determines whether a string contains the characters of a specified string.

It returns true if the string contains the value, otherwise it returns false.

Syntax : string.includes(searchvalue, start)

Example:

```
const str = "Learn 4.0 Technologies";

const word = str.includes("Tech");

const number = str.includes("5.0");

console.log(word); // true

console.log(number); // false
```

4.8 concat()

The **concat()** method is used to join two or more strings.

Syntax : string.concat(string1, string2, ..., stringX)

Example:

```
const str1 = "Hello";

const str2 = "World";

console.log(str1.concat(str2)); // HelloWorld

console.log(str1.concat(" Pavan", ". Have a nice day.")); // Hello Pavan. Have a nice day.
```

4.9 indexOf()

The **indexOf()** method returns the position of the first occurrence of a specified value in a string.

Syntax : string.indexOf(searchvalue, start)

Example:

```
const str = "Building Global Startups";

console.log(str.indexOf("Global")); // 9

console.log(str.indexOf("up")); // 21
```

4.10 startsWith()

The **startsWith()** method determines whether a string begins with the characters of a specified string, returning true or false as appropriate.

Syntax : string.startsWith(searchvalue, start)

Example:

const str = "World-class Products";

console.log(str.startsWith("rld")); // **false**

console.log(str.startsWith("World")); // **true**

4.11 endsWith()

The **endsWith()** method determines whether a string ends with the characters of a specified string, returning true or false as appropriate.

Syntax : string.endsWith(searchvalue, length)

Example:

const str = "How are you?";

console.log(str.endsWith("you?")); // **true**

console.log(str.endsWith("re")); // **false**

4.12 toString()

The **toString()** method returns the value of a string object.

Syntax : string.toString()

Example:

const number = 46;

const newNumber = number.toString();

console.log(newNumber); // **46**

console.log(typeof newNumber); // **string**

4.13 substring()

The **substring()** method returns the part of the string between the start and end indexes, or to the end of the string.

Syntax : string.substring(start, end)

Example:

const str = "I am learning JavaScript";

console.log(str.substring(2, 9)); // **am lear**

console.log(str.substring(6)); // **earning JavaScript**

4.14 Length

The **length** property returns the length of a string (number of characters).

Syntax : string.length

Example:

const str = "Upgrade to CCBP Tech 4.0 Intensive";

console.log(str.length); // **34**