JavaScript

Introduction:

- JavaScript is a dynamic computer programming language.
- It is lightweight and most commonly used as a part of web pages.
- Javascript implementations allow client-side script to interact with the user and make dynamic pages.
- It is an interpreted programming language with object-oriented capabilities.

- European Computer Manufacturers Association (ECMAScript) or (ES) is a standard for scripting languages like JavaScript, ActionScript and Jscript.

Advantages:

1. Less server interaction
2. Immediate Feedback to the visitors
3. Increased interactivity
4. Rich Interfaces

Introduction: ES5

- Data Types:
- JavaScript allows you to work with three primitive data types –
- **Numbers,** eg. 123, 120.50 etc.
- **Strings** of text e.g. "This text string" etc.
- **Boolean** e.g. true or false.
- JavaScript also defines two trivial data types, **null** and **undefined,** each of which defines only a single value.
- In addition to these primitive data types, JavaScript supports a composite data type known as **object**

Syntax:

- The console is a panel that displays important messages, like errors, for developers.
- Much of the work the computer does with our code is invisible to us by default.
- If we want to see things appear on our screen, we can print, or log, to our console directly.
- In JavaScript, the console keyword refers to an object, a collection of data and actions, that we can use in our code.
- Keywords are words that are built into the JavaScript language, so the computer will recognize them and treats them specially.
- One action, or method, that is built into the console object is the .log() method.
- When we write console.log() what we put inside the parentheses will get printed, or logged, to the console.

ES6 Syntax:

- A JavaScript program can be composed of –
- **Variables** – Represents a named memory block that can store values for the program.
- **Literals** – Represents constant/fixed values.
- **Operators** – Symbols that define how the operands will be processed.
- **Keywords** – Words that have a special meaning in the context of a language.
- **Modules** – Represents code blocks that can be reused across different programs/scripts.
- **Comments** – Used to improve code readability. These are ignored by the JavaScript engine.

- **Identifiers** − These are the names given to elements in a program like variables, functions, etc.
- The rules for identifiers are −
  - Identifiers can include both, characters and digits.
  - The identifier cannot begin with a digit.
  - Identifiers cannot include special symbols except for underscore (_) or a dollar sign ($).
  - Identifiers cannot be keywords.
  - They must be unique.
  - Identifiers are case sensitive.
  - Identifiers cannot contain spaces.

White Spaces and Line Breaks:

- ES6 ignores spaces, tabs, and newlines that appear in programs.
- JavaScript is case-sensitive.
- Each line of instruction is called a **statement**.
- Semicolons are optional in JavaScript.
- Console.log(“Hi”)

Variables:

- acts as a container for values in a program.
- Variable names are called **identifiers**.
- A variable must be declared before it is used.
- ES5 syntax used the **var** keyword to achieve the same

JavaScript and Dynamic Typing:

- JavaScript is an un-typed language.
- This means that a JavaScript variable can hold a value of any data type.
- You don't have to tell JavaScript during variable declaration what type of value the variable will hold.
- The value type of a variable can change during the execution of a program and JavaScript takes care of it automatically.

JavaScript Variable scope:

- Global Variables:
  - it can be defined anywhere in your JavaScript code.
- Local Variables:
  - visible only within a function where it is defined.
  - Function parameters are always local to that function.
- E.g.
- ES6 defines a new variable scope- The Block Scope

Function scope

- Variable having Function-scope means, variable will only be available to use inside the function it declared,
- will not be accessible outside of function,
- and will give Reference Error if we try to access.

```
function myFun() {
        var a=10;
        console.log(a);
        }

myFun();
console.log(a)
```

Block scope:

- Block means a pair of curly brackets,
- a block can be anything that contains an opening and closing curly bracket.
- Variable having Block-scope will only be available to use inside the block it declared,
- will not be accessible outside the block,
- and will give Reference Error if we try to access.

```
 <script>
        if(true) {
    let a = 10;
    console.log(a);
}
console.log(a);

        </script>
```

Var, let and const

- Var variables can be update and redeclared within its scope.
- Let variables can be updated but not redeclared
- Const variables can neither be updated nor redeclared.
- Variables declared with var are in the function scope.
- Variables declared as let are in the block scope
- Variables declared as const are in the block scope.

- Variables declared with var can be redeclared and reassigned.

```
var number = 50
```

- You have the var keyword, the name of the variable number, and an initial value **50**.
- If an initial value is not provided, the default value will be **undefined**:

```
var number
console.log(number)

// undefined
```

- The **`var`** keyword allows for redeclaration

```
var number = 50
console.log(number) // 50
var number = 100

console.log(number) // 100
```

- The **`var`** keyword also allows for reassignment

```
var number = 50
console.log(number) // 50

number = 100
console.log(number) // 100

number = 200

 console.log(number) // 200
```

Hoisting:

- Prior to ES6, the **var** keyword was used to declare a variable in JavaScript.
- Variables declared using **var** do not support block level scope.
- This means if a variable is declared in a loop or **if block** it can be accessed outside the loop or the **if block**.
- This is because the variables declared using the **var** keyword support hoisting.
- Hoisting is JavaScript's default behavior of moving declarations to the top.
- a variable can be used before it has been declared.

- **Variable hoisting** allows the use of a variable in a JavaScript program, even before it is declared.
- Such variables will be initialized to **undefined** by default.
- JavaScript runtime will scan for variable declarations and put them to the top of the function or script.
- Variables declared with **var** keyword get hoisted to the top.

Hoisting:

- Variables declared with **`var`** are hoisted to the top of their global or local scope, which makes them accessible before the line they are declared.

```
console.log(number) // undefined
var number = 50
console.log(number) // 50
```

- we can access the variable before the line where it was declared without errors.
- But the variable is hoisted with a default value of **undefined**.

Hoisting:local scope

```
function print() {
  var square1 = number * number
  console.log(square1)

  var number = 50

  var square2 = number * number
  console.log(square2)
}

print()
// NaN

  // 2500
```

Let variable:

- If we try to declare a let variable twice within the same block it will throw an error.
- But let variable can be used in different block level scopes without any syntax error.

The scope of variables declared with `let`

- Variables declared with **`let`** can have a **global**, **local**, or **block scope**.
- Block scope is for variables declared in a block.
- A block in JavaScript involves opening and closing curly braces:

```
{
  // a block

  }
```

- You can find blocks in **`if`**, **`loop`**, **`switch`**, and a couple of other statements.
- Any variables declared in such blocks with the **`let`** keyword will have a block scope.
- Also, you can't access these variables outside the block.

```
let number = 50
function print() {
  let square = number * number

  if (number < 60) {
    var largerNumber = 80
    let anotherLargerNumber = 100
    console.log(square)
  }
  console.log(largerNumber)
  console.log(anotherLargerNumber)
}
print()
// 2500
// 80

  // ReferenceError: anotherLargerNumber is not defined
```

- Just like **`var`**, variables declared with **`let`** can be reassigned to other values, but they cannot be redeclared.

```
let number = 50
console.log(number) // 50
number = 100

 console.log(number) // 100
```

```
let number = 50

let number = 100

 // SyntaxError: Identifier 'number' has already been declared
```

The Let and Block Scope:

- The block scope restricts a variable’s access to the block in which it is declared.
- The **var** keyword assigns a function scope to the variable.

The const:

- The **const** declaration creates a read-only reference to a value.
- Constants are block-scoped, much like variables defined using the let statement.
- The value of a constant cannot change through re-assignment, and it can't be re-declared.
- The following rules hold true for a variable declared using the **const** keyword −
  - Constants cannot be reassigned a value.
  - A constant cannot be re-declared.
  - A constant requires an initializer. This means constants must be initialized during its declaration.
  - The value assigned to a **const** variable is mutable.

JavaScript “use strict”

- Use strict;
- Defines that javascript should be executed in strict mode.
- The “use strict” directive was new in ES5
- It is not a statement but a literal, ignored by earlier version of Javascript.
- Its purpose is to indicate that the code should be executed in strict mode.
- With strict mode we can not use undeclared variables.
- Strict mode is declared by adding "use strict"; to the beginning of a script or a function.
- Declared at the beginning of a script, it has global scope (all code in the script will execute in strict mode)

JavaScript “use strict”

- The strict mode in JavaScript does **not** allow following things:
- Use of undefined variables
- Use of reserved keywords as variable or function name
- Duplicate properties of an object
- Duplicate parameters of function

Operator:

- An **expression** is a special kind of statement that evaluates to a value. Every expression is composed of −
- **Operands** − Represents the data.
- **Operator** − Defines how the operands will be processed to produce a value.
- JavaScript supports the following types of operators −
  - Arithmetic operators
  - Logical operators
  - Relational operators
  - Bitwise operators
  - Assignment operators
  - Ternary/conditional operators
  - String operators
  - Type operators
  - The void operator

Arithmetic Operators

- Addition
- Subtraction
- Multiplication
- Division
- Modulus
- Increment
- Decrement

Test ? expr1 : expr2

Conditional Operator:

- This operator is used to represent a conditional expression.
- The conditional operator is also sometimes referred to as the ternary operator.
- Where,
- **Test** – Refers to the conditional expression
- **expr1** – Value returned if the condition is true
- **expr2** – Value returned if the condition is false

String Operators : Concatenation operator (+)

- The + operator when applied to strings appends the second string to the first.
- The concatenation operation doesn’t add a space between the strings.
- Multiple strings can be concatenated in a single statement.

Typeof Operator

- It is a unary operator.
- This operator returns the data type of the operand.

Type	String Returned by typeof
Number	"number"
String	"string"
Boolean	"boolean"
Object	"object"

Decision Making:

Loops:

- Certain instructions require repeated execution.
- Loops are an ideal way to do the same.
- A loop represents a set of instructions that must be repeated.
- Definite Loop:
  - A loop whose number of iterations are definite/fixed is termed as a definite loop.
  - The ‘for loop’ is an implementation of a definite loop.

```
for (variablename in object) { statement or block to execute }
```

Loops: for…in loop

- The for...in loop is used to loop through an object's properties.

Spread Operator

- **ES6** provides a new operator called the **spread operator**.
- The spread operator is represented by three dots “...” .
- The spread operator converts an array into individual array elements.
- The spread operator can be used to copy one array into another.
- It can also be used to concatenate two or more arrays.

Copy Array Using Spread Operator

- You can also use the spread syntax `...` to copy the items into a single array.

```
const arr1 = ['one', 'two'];
const arr2 = [...arr1, 'three', 'four', 'five'];

console.log(arr2);
//  Output:
//  ["one", "two", "three", "four", "five"]
```

- However, if you want to copy arrays so that they do not refer to the same array, you can use the spread operator.
- This way, the change in one array is not reflected in the other.

```
let arr1 = [ 1, 2, 3];

// copy using spread syntax
let arr2 = [...arr1];

console.log(arr1); // [1, 2, 3]
console.log(arr2); // [1, 2, 3]

// append an item to the array
arr1.push(4);

console.log(arr1); // [1, 2, 3, 4]
console.log(arr2); // [1, 2, 3]
```

Clone Array Using Spread Operator

In JavaScript, objects are assigned by reference and not by values

```
let arr1 = [ 1, 2, 3];
let arr2 = arr1;

console.log(arr1); // [1, 2, 3]
console.log(arr2); // [1, 2, 3]

// append an item to the array
arr1.push(4);

console.log(arr1); // [1, 2, 3, 4]
console.log(arr2); // [1, 2, 3, 4]
```

Spread Operator using Object

```
const obj1 = { x : 1, y : 2 };
const obj2 = { z : 3 };

// add members obj1 and obj2  to obj3
const obj3 = {...obj1, ...obj2};

console.log(obj3); // {x: 1, y: 2, z: 3}
```

Functions:

- **Functions** are the building blocks of readable, maintainable, and reusable code.
- Functions are defined using the function keyword.
- To force execution of the function, it must be called.
- This is called as function invocation

Classification of function:

1. Returning
2. Parametrized

- Functions may also return the value along with control, back to the caller.
- Such functions are called as returning functions.
- A returning function must end with a return statement.
- A function can return at the most one value. In other words, there can be only one return statement per function.
- The return statement should be the last statement in the function.

- Parameters are a mechanism to pass values to functions.
- Parameters form a part of the function’s signature.
- The parameter values are passed to the function during its invocation.
- Unless explicitly specified, the number of values passed to a function must match the number of parameters defined.

Default Function Parameter

- In ES6, a function allows the parameters to be initialized with default values, if no values are passed to it or it is undefined.

```
function sum(x = 3, y = 5) {

    // return sum
    return x + y;
}

console.log(sum(5, 15));  // 20
console.log(sum(7));      // 12
console.log(sum());       // 8
```

Using Expression as default value

- It is also possible to provide expressions as default values.

```
function sum(x = 1, y = x,  z = x + y) {
    console.log( x + y + z );
}

sum(); // 4
```

- If you reference the parameter that has not been initialized yet, you will get an error.

```
function sum( x = y, y = 1 ) {
    console.log( x + y);
}
sum();
```

```
ReferenceError: Cannot access 'y' before initialization
```

Passing Function Value as Default Value

```
// using a function in default value expression

const sum = () => 15;

const calculate = function( x, y = x * sum() ) {
    return x + y;
}

const result = calculate(10);
console.log(result);            // 160
```

Passing undefined value

- In JavaScript, when you pass `undefined` to a default parameter function, the function takes the default value.

```
function test(x = 1) {
  console.log(x);
}
// passing undefined
// takes default value 1
test(undefined); // 1
```

- Rest Parameters:
- Rest parameters are similar to variable arguments in Java.
- Rest parameters doesn’t restrict the number of values that you can pass to a function.
- To declare a rest parameter, the parameter name is prefixed with three periods, known as the spread operator.
- A rest parameter allows you to represent an indefinite number of arguments as an array.

```
let func = function(...args) {
    console.log(args);
}

func(3); // [3]
func(4, 5, 6); // [4, 5, 6]
```

- You can also accept multiple arguments in a function call using the rest parameter.

- You can also pass multiple arguments to a function using the spread operator.

```
function sum(x, y ,z) {
    console.log(x + y + z);
}

const num1 = [1, 3, 4, 5];

sum(...num1); // 8
```

- If you pass multiple arguments using the spread operator, the function takes the required arguments and ignores the rest.

Anonymous Function

- Functions that are not bound to an identifier (function name) are called as anonymous functions.
- These functions are dynamically declared at runtime.
- Anonymous functions can accept inputs and return outputs, just as standard functions do.
- An anonymous function is usually not accessible after its initial creation.
- Variables can be assigned an anonymous function.
- Such an expression is called a **function expression**.

- Syntax: var res = function([arguments]) { ... }

```
var f = function(){ return "hello"}
console.log(f())
```

Function Constructor:

- We can define function dynamically using Function() constructor along with the new operator.

- var variablename = new Function(Arg1, Arg2..., "Function Body");
- The Function() constructor expects any number of string arguments.
- The last argument is the body of the function
- It can contain arbitrary JavaScript statements, separated from each other by semicolons.

```
var func = new Function("x", "y", "return x*y;");
        function secondFunction() {
           var result;
           result = func(10,20);
           document.write ( result );
```

- Recursion:
- Recursion is a technique for iterating over an operation by having a function call itself repeatedly until it arrives at a result.
- Recursion is best applied when you need to call the same function repeatedly with different parameters from within a loop.

Lambda Function/ Arrow Function

- refers to anonymous functions in programming.
- Lambda functions are a concise mechanism to represent anonymous functions.
- These functions are also called as Arrow functions.
- There are 3 parts to a Lambda function −
  - Parameters − A function may optionally have parameters.
  - The fat arrow notation/lambda notation (=>): It is also called as the goes to operator.
  - Statements − Represents the function's instruction set.

```
([param1, parma2,…param n] )=>statement;
```

- Arrow functions remove the need to type out the keyword function every time you need to create a function.
- Instead, you first include the parameters inside the ( ) and then add an arrow => that points to the function body surrounded in { }.

- Lambda Expression:
- It is an anonymous function expression that points to a single line of code.
- It is a compact alternative to a traditional function expression, but is limited and can't be used in all situations.

```
let x = function(x, y) {
   return x * y;
 }
```

can be written as

```
 let x = (x, y) => x * y;  //Arrow Function
```

Arrow function with no argument

```
let greet = () => console.log('Hello');
greet(); // Hello
```

If a function doesn't take any argument, then you should use empty parentheses.

Arrow function with one argument

```
let greet = x => console.log(x);
greet('Hello'); // Hello
```

If a function has only one argument, you can omit the parentheses.

Arrow function as an expression

```
let age = 5;

let welcome = (age < 18) ?
  () => console.log(‘Not Eligible') :
  () => console.log('Eligible for Voting');

welcome();
```

Multiline Arrow functions

```
let sum = (a, b) => {
    let result = a + b;
    return result;
}

let result1 = sum(5,7);
console.log(result1); // 12
```

Argument binding

- Regular functions have arguments binding.
- That's why when you pass arguments to a regular function, you can access them using the `arguments` keyword.

```
let x = function () {
    console.log(arguments);
}
x(4,6,7); // Arguments [4, 6, 7]
```

- Arrow functions do not have arguments binding.
- When you try to access an argument using the arrow function, it will give an error

```
let x = () => {
    console.log(arguments);
}
x(4,6,7);
 // ReferenceError: Can't find variable: arguments
```

```
let x = (...n) => {
    console.log(n);
}

 x(4,6,7); // [4, 6, 7]
```

Callback Function

- A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

Syntax:

Arr.forEach(callback(current value [, index [, array]])[, thisArgs])

Parameters:

- The forEach method passes a callback function for each element of an array together with the following parameters:
- Current Value (required) - The value of the current array element
- Index (optional) - The current element's index number
- Array (optional) - The array object to which the current element belongs

Iterator:

- JavaScript iterators were introduced in ES6
- They are used to loop over a sequence of values.
- An iterator implements a next() function, that returns an object in the form of { value, done }
- where value is the next value in the iteration sequence
- and done is a boolean determining if the sequence has already been consumed.
- For
- For…in
- For …of
- forEach(callback())

```
let ranks = ['A', 'B', 'C'];

for (let i = 0; i < ranks.length; i++) {
    console.log(ranks[i]);
}
```

- The for loop uses the variable i to track the index of the ranks array.
- The value of i increments each time the loop executes as long as the value of i is less than the number of elements in the ranks array.
- The code complexity grows when we nest a loop inside another loop.
- Keeping track of multiple variables inside the loops is error-prone.
- ES6 introduced a new loop construct called for...of to eliminate the standard loop’s complexity
- and avoid the errors caused by keeping track of loop indexes.

- To iterate over the elements of the ranks array, use the following for...of construct:

```
for(let rank of ranks) {
    console.log(rank);
}
```

- For…in loop allows you to iterate overall property keys of an object.

```
for (index in number)
{
        console.log (index);
}
```

```
let number=[10,20,30,40,50];
for (let i=0;i<number.length;i++)
console.log(number[i])

let number=[10,20,30,40,50];
for (let i=0;i<number.length;i++)
console.log(i);

for(index in number)
console.log(index);

for (index of number)
console.log(index)
```

- forEach():
- The forEach() method calls a function once for each element in an array in order
- The forEach() method passes a callback function for each element of an array together.
- The forEach() array method loops through any array, executing a provided function once for each array element in ascending index order.
- This function is referred to as a callback function.

Arr.forEach(callback(current value [, index [, array]])[, thisvalue])

- callbackFunction: The callback function is a function that is executed only once for each element
- It can accepts the following arguments to be used within the callback function.
- currentElement: The current element, as the name implies, is the element in the array that is being processed at the time the loop occurs. It is the only necessary argument.
- index: index is an optional argument that carries the index of the currentElement.
- array: The array is an optional argument that returns the array that was passed to the forEach() method.
- thisValue: This is an optional parameter that specifies the value that will be used in the callback function.

```
const staffsDetails = [
    { name: "Swati", age: 44, salary: 40000, currency: "Rupees" },
    { name: "Ahana", age: 34, salary: 30000, currency: "Rupees" },
    { name: "Rushda", age: 37, salary: 37000, currency: "Rupees" }

  ];
  staffsDetails.forEach((staffDetail) => {
    let sentence = `I am ${staffDetail.name} a staff of TSEC.`;
    console.log(sentence);
  });
```

Iterator

- Iterator is an object which allows us to access a collection of objects one at a time.
- The following built-in types are by default iterable –
  - String
  - Array
  - Map
  - Set
- An object is considered **iterable**, if the object implements a function whose key is **[Symbol.iterator]** and returns an iterator.
- A for...of loop can be used to iterate a collection.

- JavaScript iterators were introduced in ES6
- They are used to loop over a sequence of values, usually some sort of collection.
- An iterator must implement a next() function, that returns an object in the form of { value, done } where value is the next value in the iteration sequence and done is a boolean determining if the sequence has already been consumed.

```
<script>
let marks = [10,20,30]
//check iterable using for..of
for(let m of marks){
console.log(m);
}
</script>
```

```
<script>
let marks = [10,20,30]
let iter = marks[Symbol.iterator]();
console.log(iter.next())
console.log(iter.next())
console.log(iter.next())
console.log(iter.next())
</script>
```

```
{value: 10, done: false}
{value: 20, done: false}
{value: 30, done: false}
{value: undefined, done: true}
```

JavaScript next() Method

The iterator object has a `next()` method that returns the next item in the sequence.

The `next()` method contains two properties: `value` and `done`.

- value
  The `value` property can be of any data type and represents the current value in the sequence.
- done
  The `done` property is a boolean value that indicates whether the iteration is complete or not. If the iteration is incomplete, the `done` property is set to `false`, else it is set to `true`.

```
const arr = ['h', 'e', 'l', 'l', 'o'];

let arrIterator = arr[Symbol.iterator]();

console.log(arrIterator.next()); // {value: "h", done: false}
console.log(arrIterator.next()); // {value: "e", done: false}
console.log(arrIterator.next()); // {value: "l", done: false}
console.log(arrIterator.next()); // {value: "l", done: false}
console.log(arrIterator.next()); // {value: "o", done: false}
console.log(arrIterator.next()); // {value: undefined, done: true}
```

Generator:

- Prior to ES6, functions in JavaScript followed a run-to completion model.
- ES6 introduces functions known as Generator which can stop midway and then continue from where it stopped.
- A generator prefixes the function name with an asterisk * character and contains one or more **yield** statements.
- The **yield** keyword returns an iterator object.

Create JavaScript Generators

- To create a generator, you need to first define a generator function with `function*` symbol.
- The objects of generator functions are called generators.
- The generator function is denoted by *.

```
// define a generator function
function* generator_function() {
    ... .. ...
}

// creating a generator
 const generator_obj = generator_function();
```

Using yield statement to pause Execution

- We can pause the execution of a generator function without executing the whole function body.
- For that, we use the `yield` keyword.

```
// generator function
function* generatorFunc() {

    console.log("1. code before the first yield");
    yield 100;

   console.log("2. code before the second yield");
    yield 200;
}

// returns generator object
const generator = generatorFunc();

console.log(generator.next());
```

```
1. code before the first yield
 {value: 100, done: false}
```

- When `generator.next()` is called, the code up to the first `yield` is executed.
- When `yield` is encountered, the program returns the value and pauses the generator function.
- yield does not terminate the program like return statement.
- We can continue executing code from the last yielded position.

```
function* generatorFunc() {

    console.log("1. code before first yield");
    yield 100;

   console.log("2. code before the second yield");
    yield 200;

    console.log("3. code after the second yield");
}

const generator = generatorFunc();

console.log(generator.next());
console.log(generator.next());
console.log(generator.next());
```

```
1. code before first yield
{value: 100, done: false}
2. code before second yield
{value: 200, done: false}
 {value: undefined, done: true}
```

```
<script>
//define generator function
function * getMarks(){
console.log("Step 1")
yield 10
console.log("Step 2")
yield 20
console.log("Step 3")
yield 30
console.log("End of function")
}
//return an iterator object
let markIter = getMarks()
```

```
//invoke statements until first yield
console.log(markIter.next())

//resume execution after the last yield until
second yield expression
console.log(markIter.next())

//resume execution after last yield until third
yield expression
console.log(markIter.next())

console.log(markIter.next())
// iteration is completed;no value is returned
</script>
```

We can also pass an argument to a generator function.

```
// generator function
function* generatorFunc() {

    // returns 'hello' at first next()
    let x = yield 'hello';

    // returns passed argument on the second next()
    console.log(x);
    console.log('some code');

    // returns 5 on second next()
    yield 5;

}

const generator = generatorFunc();

console.log(generator.next());
console.log(generator.next(6));
console.log(generator.next());
```

```
{value: "hello", done: false}
6
some code
{value: 5, done: false}
 {value: undefined, done: true}
```

- We can use the `return` statement in a generator function.
- The `return` statement returns a value and terminates the function.

```
// generator function
function* generatorFunc() {

    yield 100;

   return 123;

   console.log("2. some code before second yield");
    yield 200;
}

// returns generator object
const generator = generatorFunc();

console.log(generator.next());
console.log(generator.next());
console.log(generator.next());
```

```
{value: 100, done: false}
{value: 123, done: true}
{value: undefined, done: true}
```

Uses of Generators:

- Generators provide an easier way to implement iterators.
- Generators execute its code only when required.
- Generators are memory efficient.

Javascript Events:

- An event is something that happens when user interact with the web page, such as when
- he clicked a link or button,
- entered text into an input box or textarea,
- made selection in a select box,
- pressed key on the keyboard,
- moved the mouse pointer,
- submits a form, etc.
- Sometime the Browser itself can trigger the events, such as the page load and unload events.

- When an event occur, you can use a JavaScript event handler (or an event listener) to detect them and perform specific task or set of tasks.
- The names for event handlers always begin with the word "on",
- Event handler for the click event is called `onclick`

- The events can be categorized into four main groups
  - mouse events,
  - keyboard events,
  - form events and
  - document/window events.

Mouse Events:

- A mouse event is triggered when the user click some element, move the mouse pointer over an element.
- e.g. onclick, onmouseover, onmouseout

```
<button type="button" id="myBtn">Click Me</button>
  <script>
     function sayHello(){
        alert('Hello World!');
     }
     document.getElementById("myBtn").onclick = sayHello;
  </script>
```

Keyboard Events:

- A keyboard event is fired when the user press or release a key on the keyboard.
- e.g. onkeydown, onkeyup, onkeypress
- The keydown event occurs when the user presses down a key on the keyboard.
- The keyup event occurs when the user releases a key on the keyboard.
- The keypress event occurs when a user presses down a key on the keyboard that has a character value associated with it.
- e.g keys like Ctrl, Shift, Alt, Esc, Arrow keys, etc. will not generate a keypress event, but will generate a keydown and keyup event.

Form Events:

- A form event is fired when a form control receive or loses focus or when the user modify a form control value.
- e.g by typing text in a text input, select any option in a select box etc.
- onfocus, onblur, onsubmit, onchange

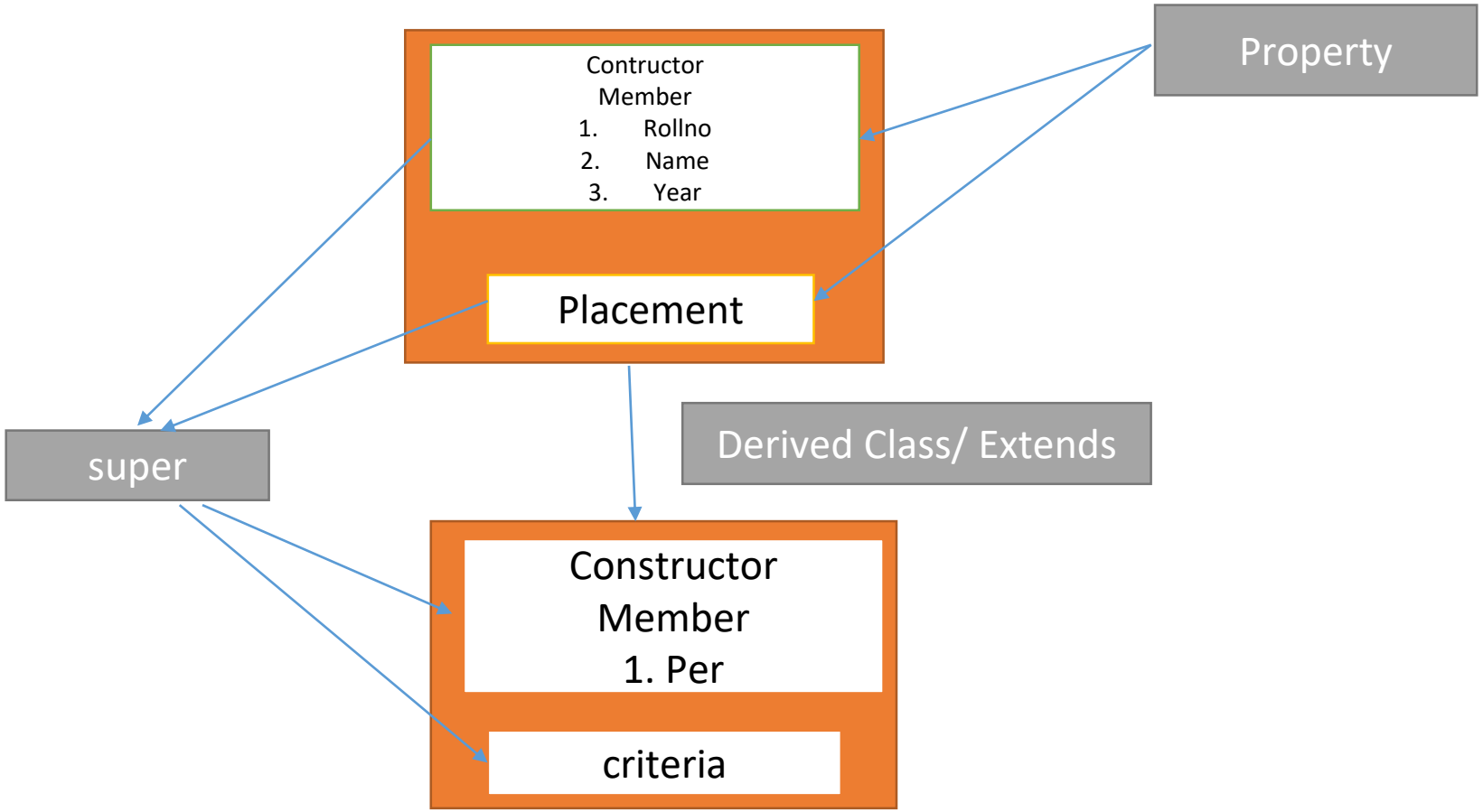
```
<form action="action.php" method="post" onsubmit="alert('Form data will be submitted to
the server!');">
<label>First Name:</label>
<input type="text" name="first-name" required>
<input type="submit" value="Submit">
 </form>
```

Document / Window Events:

- Events are also triggered in situations when the page has loaded or when user resize the browser window.
- **The Load Event (onload)**
- The load event occurs when a web page has finished loading in the web browser.
- **The Unload Event (onunload)**
- The unload event occurs when a user leaves the current web page.
- **The Resize Event (onresize)**
- The resize event occurs when a user resizes the browser window. The resize event also occurs in situations when the browser window is minimized or maximized.

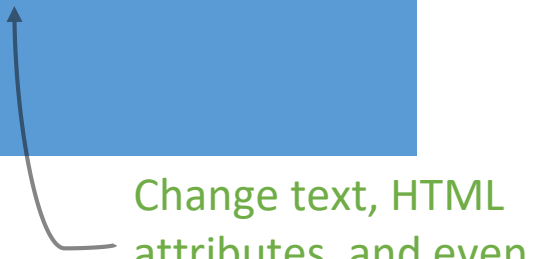
Classes and Inheritance:

- Classes: In OO programming, a class is blueprint for creating objects, providing initial values for state and implementations of behavior
- Constructor: The constructor is called on an object after it has been created and is a good place to put initialization code.
- Property: is a special sort of class member, intermediate in functionality between a field and a method
- Object: Each object is an instance of a particular class or subclass with the class own methods or procedures and data variables.
- Extends: For inheritance
- Super: to access member or property and method from parent class
- Instance: can pass member to derived class or child class
- Static: can not pass member to derived class or child class


Contructor
Member
1. Rollno
2. Name
3. Year
Property
Placement
super
Derived Class/ Extends
Constructor
Member
1. Per
criteria

What is DOM?

DOCUMENT OBJECT MODEL: STRUCTURED REPRESENTATION OF HTML DOCUMENTS. ALLOWS JAVASCRIPT TO ACCESS HTML ELEMENTS AND STYLES TO MANIPULATE THEM.



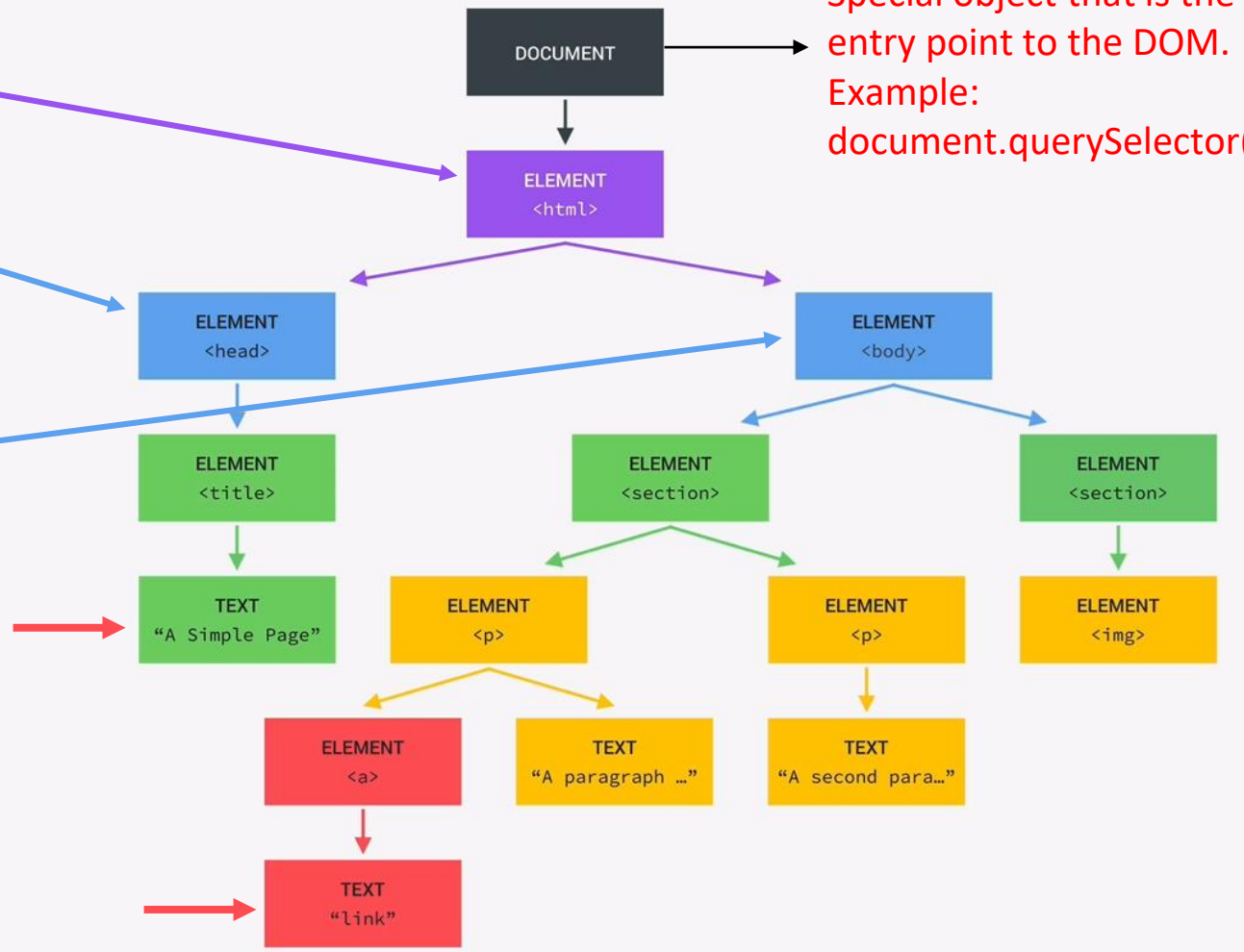
Change text, HTML attributes, and even CSS styles.

DOM Manipulation:

- The Document Object Model (DOM) is an application programming interface (API) for manipulating HTML and XML documents.
- The DOM represents a document as a tree of nodes.
- It provides API that allows you to add, remove, and modify parts of the document effectively.
- In DOM tree, the document is the root node.
- The root node has one child which is the <html> element.
- The <html> element is called document element.
- Each document can have only one document element.
- In an HTML document, the document element is the <html>.
- Each markup can be represented by a node in the tree.

THE DOM TREE STRUCTURE

```
<html>
  <head>
    <title>A Simple Page</title>
  </head>
  <body>
    <section>
      <p>A paragraph with a <a>link</a></p>
      <p>A second paragraph</p>
    </section>
    <section>
      <img src="dom.png" alt="The DOM" />
    </section>
  </body>
</html>
```



Special object that is the entry point to the DOM. Example: document.querySelector().

- There are different types of DOM supported by javascript.

1. Legacy DOM:

- This was the model used by early versions of JavaScript.
- This model provides read-only properties such as title, URL, etc.
- It also provides with lastModified information about the document as a whole.
- This model has a lot of methods that can be used to set and get the document property value.

Document Properties of Legacy DOM

- alinkcolor: this property defines color of activated links.
- document.alinkcolor
- vlinkcolor: this property defines color of visited links.
- document.vlinkcolor
- linkcolor: this property defines color of unvisited links.
- document.linkcolor
- Title:contents of title tag
- document.title
- Fgcolor:defines the default text color of the document
- Document.fgcolor
- Bgcolor:defines background color of the document

W3C DOM:

- With the object model, JavaScript gets all the power it needs to create dynamic HTML:
- JavaScript can change all the HTML elements in the page
- JavaScript can change all the HTML attributes in the page
- JavaScript can change all the CSS styles in the page
- JavaScript can remove existing HTML elements and attributes
- JavaScript can add new HTML elements and attributes
- JavaScript can react to all existing HTML events in the page
- JavaScript can create new HTML events in the page

- It is the process of interacting with the DOM API to change or modify an HTML document that will be displayed in a web browser.
- This HTML document can be changed to add or remove elements, update existing elements, rearrange existing elements, etc.

Adding New Elements to the DOM

- We can explicitly create new element in an HTML document, using the `document.createElement()` method.
- This method creates a new element, but it doesn't add it to the DOM;

```
<div id="main">
<h1 id="title">Hello World!</h1>
<p id="hint">This is a simple paragraph.</p>
</div>

<script>
var newDiv = document.createElement("div");
var newContent = document.createTextNode("Hi, how are you
doing?");
newDiv.appendChild(newContent);
var currentDiv = document.getElementById("main");
document.body.appendChild(newDiv, currentDiv); </script>
```

Getting or Setting HTML Contents to DOM

- You can also get or set the contents of the HTML elements easily with the `innerHTML` property.
- This property sets or gets the HTML markup contained within the element i.e. content between its opening and closing tags.

```
 <div id="main">
		<h1 id="title">Hello World!</h1>
		<p id="hint">This is a simple paragraph.</p>
	</div>

	<button type="button" onclick="getContents()">Get Contents</button>
	<button type="button" onclick="setContents()">Set Contents</button>

	<script>
	function getContents() {
		var contents = document.getElementById("main").innerHTML;
		alert(contents);
	}
	function setContents() {
		var mainDiv = document.getElementById("main");
		mainDiv.innerHTML = "<p>This is <em>newly inserted</em>
paragraph.</p>";
	}
	</script>
```

- The `innerHTML` property replaces all existing content of an element.
- So if you want to insert the HTML into the document without replacing the existing contents of an element, you can use the `insertAdjacentHTML()` method.
- This method accepts two parameters:
  - the position in which to insert and
  - the HTML text to insert.

The position must be one of the following values:

`"beforebegin"`, `"afterbegin"`, `"beforeend"`, and `"afterend"`

```
<body>
	<div id="main">
		<h1 id="title">Hello World!</h1>
	</div>
	<button type="button" onclick="insertContent()">Insert Content</button>
	<script>
	function insertContent() {
		var mainDiv = document.getElementById("main");

		mainDiv.insertAdjacentHTML('beforebegin', '<p>This is paragraph one.</p>');

		mainDiv.insertAdjacentHTML('afterbegin', '<p>This is paragraph two.</p>');

		mainDiv.insertAdjacentHTML('beforeend', '<p>This is paragraph three.</p>');

		mainDiv.insertAdjacentHTML('afterend', '<p>This is paragraph four.</p>');
	}
	</script>
```

Remove Existing Element from DOM

- The `removeChild()` method is used to remove a child node from the DOM.

```
<div id="main">
<h1 id="title">Hello World!</h1>
<p id="hint">This is a simple paragraph.</p>
</div>
<script>
var parentElem = document.getElementById("main");
var childElem = document.getElementById("hint");
parentElem.removeChild(childElem);
</script>
```

Replace Existing Element from DOM

- Replace an element in HTML DOM with another using the `replaceChild()` method.
- This method accepts two parameters: the node to insert and the node to be replaced.
- It has the syntax like `parentNode.replaceChild(newChild, oldChild);`.

getElementById()

- The `getElementById()` method of the `Document` interface returns an `Element` object representing the element whose `id` property matches the specified string.
- Since element IDs are required to be unique if specified, they're a useful way to get access to a specific element quickly.

getElementByClassName()

- **The `getElementsByClassName` method of `Document` interface returns an array-like object of all child elements which have all of the given class name(s).**

```
var element = document.querySelector("< CSS selector >");
```

querySelector() and querySelectorAll()

- The querySelector() function takes an argument, and this argument is a string that represents the CSS selector for the element you wish to find.
- querySelector() returns the first element it finds - even if other elements exist that could get targeted by the selector.

Setting CSS styles using Javascript

querySelector()

The `querySelector()` method returns the first element that matches a CSS selector.

```
<body>

<p>This is a p element.</p>

<p>This is also a p element.</p>

<p>One more</p>

<button onclick="myFunction()">Try it</button>

<script>
function myFunction() {
    document.querySelector("p").style.backgroundColor = "red";
}
</script>
```

- querySelectorAll()
- The querySelectorAll() method returns all elements in the document that matches a specified CSS selector(s), as a static NodeList object.
- The NodeList object represents a collection of nodes.
- The nodes can be accessed by index numbers.
- The index starts at 0.

```
<body>

<p>This is a p element.</p>

<p>This is also a p element.</p>

<p>One more</p>

<button onclick="myFunction()">Try it</button>

<script>
function myFunction() {
    var x = document.querySelectorAll("p");
    x[0].style.backgroundColor = "red";
}
</script>

</body>
```

- Promises are important building blocks for asynchronous operations in JavaScript.
- A Promise is a special JavaScript object.
- It produces a value after an asynchronous operation completes successfully, or an error if it does not complete successfully due to time out, network error, and so on.
- Successful call completions are indicated by the resolve function call, and errors are indicated by the reject function call.

create a promise using the promise constructor:

```
let promise = new Promise(function(resolve, reject) {
    // Make an asynchronous call and either resolve or reject

 });
```

The constructor function takes a function as an argument.

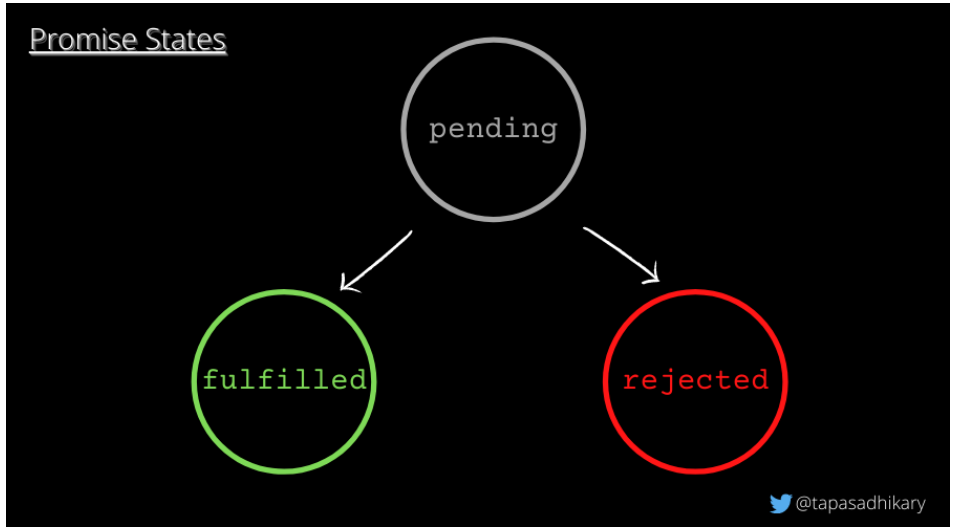
This function is called the executor function.

- The executor function takes two arguments, resolve and reject.
- These are the callbacks provided by the JavaScript language.
- For the promise to be effective, the executor function should call either of the callback functions, resolve or reject.
- The new Promise() constructor returns a promise object.
- As the executor function needs to handle async operations, the returned promise object should be capable of informing when the execution has been started, completed (resolved) or returned with error (rejected).

A promise object has the following internal properties:

1. state :This property can have the following values:

- pending: Initially when the executor function starts the execution.
- fulfilled: When the promise is resolved.
- rejected: When the promise is rejected.

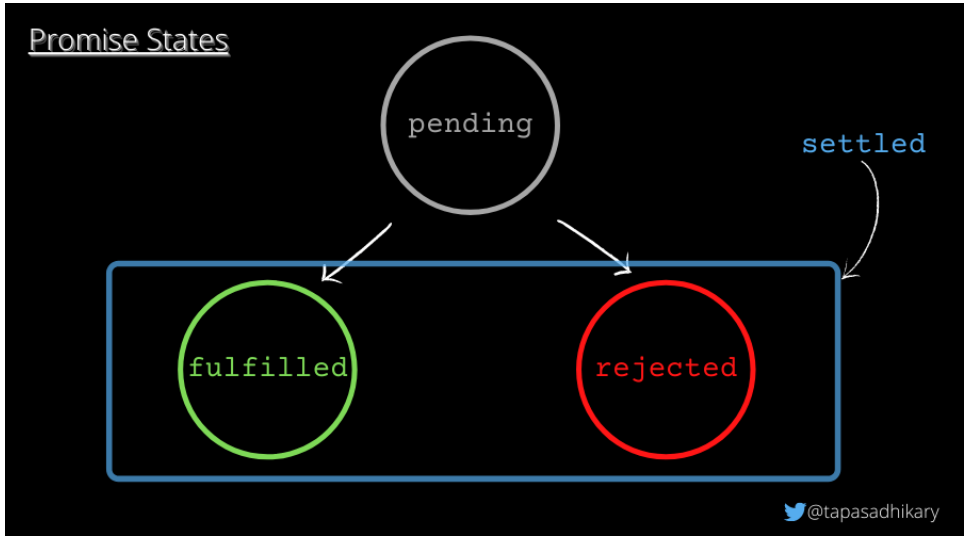


2. result – This property can have the following values:

- undefined: Initially when the state value is pending.
- value: When resolve(value) is called.
- error: When reject(error) is called.

- These internal properties are code-inaccessible but they are inspectable.
- This means that we will be able to inspect the **`state`** and **`result`** property values using the debugger tool, but we will not be able to access them directly using the program.

- A promise's state can be pending, fulfilled or rejected.
- A promise that is either resolved or rejected is called settled



- A Promise executor should call only one resolve or one reject.
- Once one state is changed (pending => fulfilled or pending => rejected).
- Any further calls to resolve or reject will be ignored.

```
let promise = new Promise(function(resolve, reject)
{
    reject(new Error('Error!'));

  });
```

```
let promise = new Promise(function(resolve, reject) {
    resolve("I am done");

  });
```

```
let promise = new Promise(function(resolve, reject) {
  resolve("resolved!");

  reject(new Error('Error')); // ignored
  resolve("Ignored?"); // ignored

 });
```

- If you are interested only in successful outcomes, you can just pass one argument to it

```
promise.then(
  (result) => {
      console.log(result);
  }

 );
```

- If you are interested only in the error outcome, you can pass **`null`** for the first argument,

```
promise.then(
  null,
  (error) => {
      console.log(
error)
  }

  );
```

- You can use the .catch() handler method to handle errors from promises.
- The syntax of passing null as the first argument to the .then() is not a great way to handle errors.
- .catch() to do the same job

- The .finally() handler performs cleanups like stopping a loader, closing a live connection, and so on.
- The finally() method will be called irrespective of whether a promise resolves or rejects.
- It passes through the result or error to the next handler which can call a .then() or .catch() again.