# Module2-Day-6

# Topics to be Covered:

Inheritance
Exceptions
Sync,asynchronous,
Promises
Recursion

# Class Inheritance

Inheritance enables you to define a class that takes all the functionality from a parent class and allows you to add more.

Using class inheritance, a class can inherit all the methods and properties of another class.

Inheritance is a useful feature that allows code reusability.

To use class inheritance, you use the `extends` keyword.

In the above example, the `Student` class inherits all the methods and properties of the `Person` class. Hence, the `Student` class will now have the `name` property and the `greet()` method.

**Output**

```
Hello Jack
```

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

    greet() {
        console.log(`Hello ${this.name}`);
    }
}

// inheriting parent class
class Student extends Person {

}

let student1 = new Student('Jack');
student1.greet();
```

# Uses of Inheritance

- Since a child class can inherit all the functionalities of the parent's class, this allows code reusability.
- Once a functionality is developed, you can simply inherit it. No need to reinvent the wheel. This allows for cleaner code and easier to maintain.
- Since you can also add your own functionalities in the child class, you can inherit only the useful functionalities and define other required features.

```
// parent class
class Person {
    constructor(name) {
        this.name = name;
        this.occupation = "unemployed";
    }

    greet() {
        console.log(`Hello ${this.name}.`);
    }

}

// inheriting parent class
class Student extends Person {

    constructor(name) {

        // call the super class constructor and pass in the name parameter
        super(name);

        // Overriding an occupation property
        this.occupation = 'Student';
    }

    // overriding Person's method
    greet() {
        console.log(`Hello student ${this.name}.`);
        console.log('occupation: ' + this.occupation);
    }
}

let p = new Student('Jack');
p.greet();
```

Here, the `occupation` property and the `greet()` method are present in parent `Person` class and the child `Student` class. Hence, the `Student` class overrides the `occupation` property and the `greet()` method.

**Output**

```
Hello student Jack.
occupation: Student
```

# Exceptions

When executing JavaScript code, errors will definitely occur. These errors can occur due to a fault from the programmer's side or the input is wrong or even if there is a problem with the logic of the program.

## Types of Errors

1. **Syntax Error:** When a user makes a mistake in the pre-defined syntax of a programming language, a syntax error may appear.
2. **Runtime Error:** When an error occurs during the execution of the program, such an error is known as Runtime error. The codes which create runtime errors are known as Exceptions. Thus, exception handlers are used for handling runtime errors.
3. **Logical Error:** An error which occurs when there is any logical mistake in the program that may not produce the desired output, and may terminate abnormally. Such an error is known as Logical error.

But all errors can be solved and to do so we use five statements that will now be explained.

The **try** statement lets you test a block of code to check for errors.

The **catch** statement lets you handle the error if any are present.

The **throw** statement lets you make your own errors.

The **finally** statement lets you execute code after try and catch.

The **finally block** runs regardless of the result of the try-catch block.

## Syntax:

```
try{
expression; } //code to be written.
catch(error){
expression; } // code for handling the error.
```

```
function geekFunc() {
        let a = 10;
        try {
console.log("Value of variable a is : " + a);
        }
        catch (e) {
console.log("Error: " + e.description);
        }
}
geekFunc();
```

# JavaScript Throws Errors

When an error occurs, JavaScript will normally stop and generate an error message.

The technical term for this is: JavaScript will throw an exception (throw an error).

**The throw Statement**

The **throw** statement allows you to create a custom error.

Technically you can **throw an exception** (**throw** an **error**).

The exception can be a JavaScript **String**, a **Number**, a **Boolean** or **an Object**:

The syntax of `try...catch...throw` is:

```
try {
    // body of try
    throw exception;
}
catch(error) {
    // body of catch
}
```

```
const number = 40;
try {
    if(number > 50) {
        console.log('Success');
    }
    else {

        // user-defined throw statement
        throw new Error('The number is low');
    }

    // if throw executes, the below code does not execute
    console.log('hello');
}
catch(error) {
    console.log('An error caught');
    console.log('Error message: ' + error);
}
```

# JavaScript try...catch...finally Statement

You can also use the `try...catch...finally` statement to handle exceptions. The `finally` block executes both when the code runs successfully or if an error occurs.

The syntax of `try...catch...finally` block is:

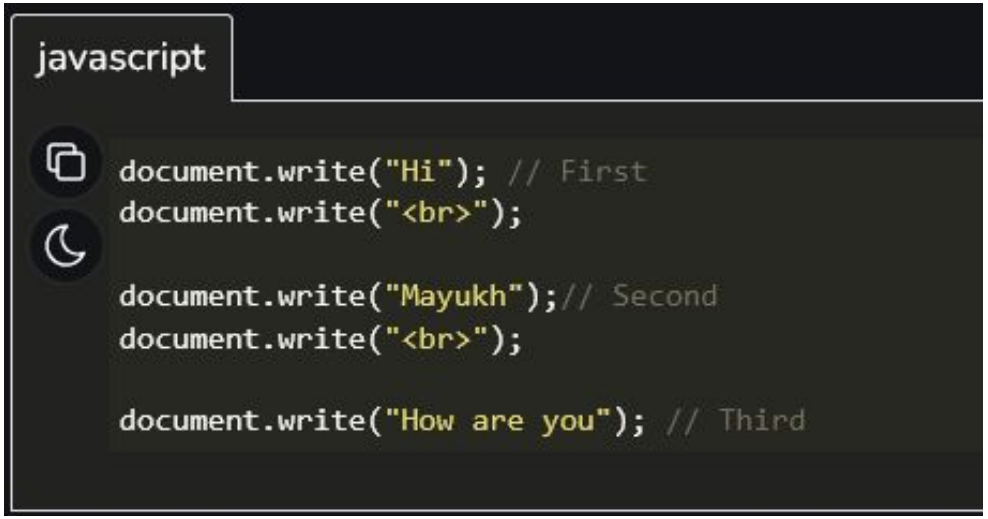
```
try {
    // try_statements
}
catch(error) {
    // catch_statements
}
finally() {
    // codes that gets executed anyway
}
```

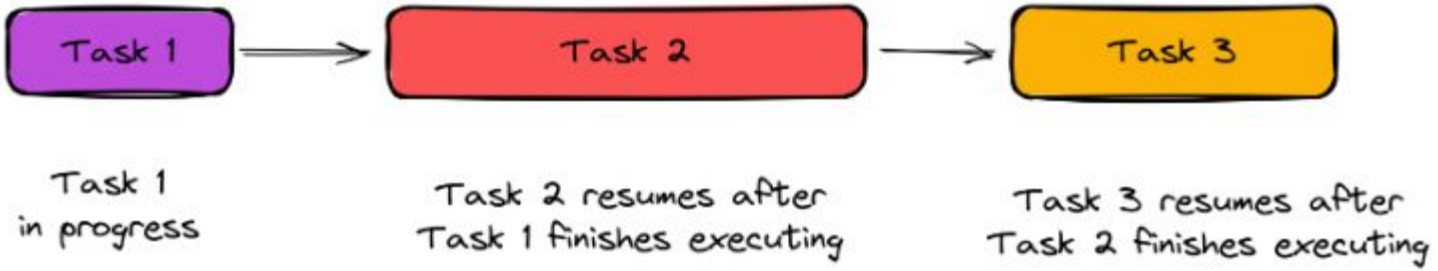
```
const numerator= 100, denominator = 'a';
try {
    console.log(numerator/denominator);
    console.log(a);}
catch(error) {
   console.log('An error caught');
   console.log('Error message: ' + error);  }
finally {
    console.log('Finally will execute every time');
}
```

# Synchronous and Asynchronous in JavaScript

**Synchronous JavaScript**

As the name suggests synchronous means to be in a sequence, i.e. every statement of the code gets executed one by one. So, basically a statement has to wait for the earlier statement to get executed.





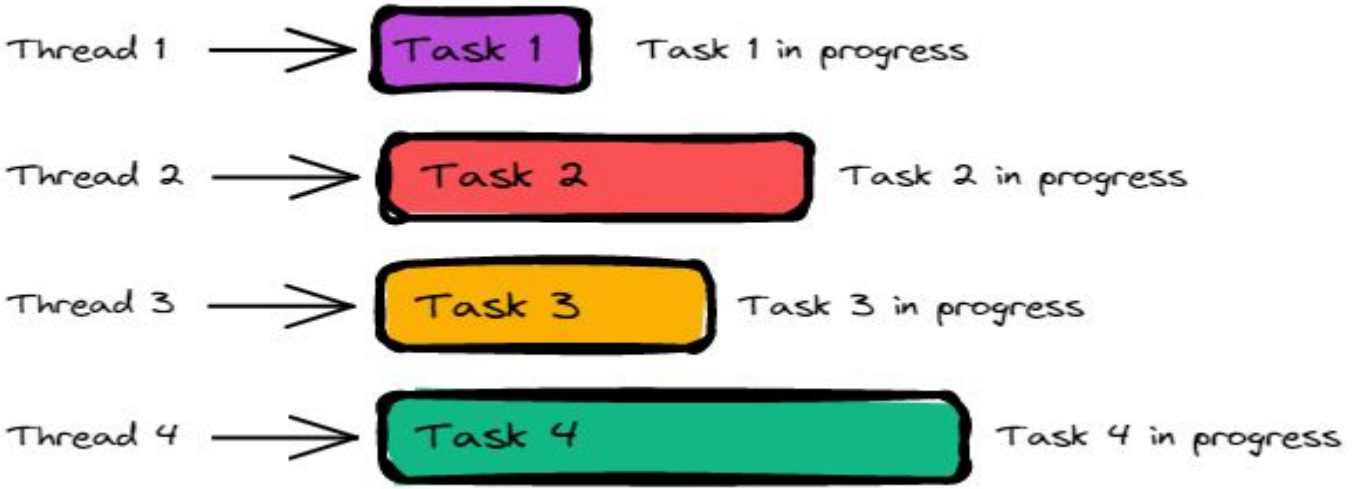

However, synchronous programming can be problematic in certain situations, particularly when dealing with tasks that take a significant amount of time to complete.

For example, let's say that a synchronous program performs a task that requires waiting for a response from a remote server. The program will be stuck waiting for the response and cannot do anything else until the response is returned. This is known as *blocking*, and it can lead to an application appearing unresponsive or **"frozen"** to the user.

# What is Asynchronous Programming?

Asynchronous code allows the program to be executed immediately whereas the synchronous code will block further execution of the remaining code until it finishes the current one. This may not look like a big problem but when you see it in a bigger picture you realize that it may lead to delaying the User Interface.

Asynchronous programming is a way for a computer program to handle multiple tasks simultaneously rather than executing them one after the other.




Asynchronous programming allows a program to continue working on other tasks while waiting for external events, such as network requests, to occur. This approach can greatly improve the performance and responsiveness of a program.

For example, while a program retrieves data from a remote server, it can continue to execute other tasks such as responding to user inputs.

```
console.log("Start of script");

setTimeout(function() {
  console.log("First timeout completed");
}, 2000);

  console.log("End of script");
```

```
Start of script
End of script
First timeout completed
```

In this example, the `setTimeout` method executes a function after a specified time. The function passed to `setTimeout` will be executed asynchronously, which means that the program will continue to execute the next line of code without waiting for the timeout to complete.

As you can see, `console.log("First timeout completed")` will be executed after 2 seconds. Meanwhile, the script continues to execute the next code statement and doesn't cause any **"blocking"** or **"freezing"** behaviour.

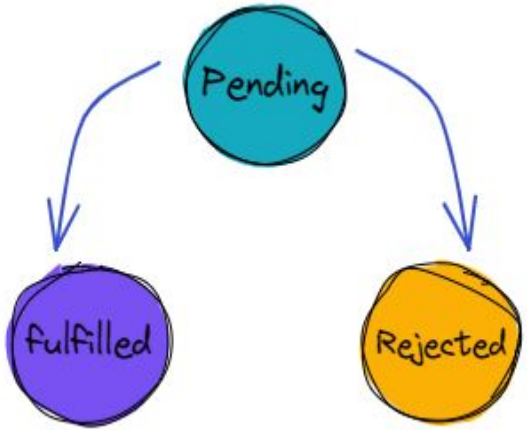
In JavaScript, asynchronous programming can be achieved through a variety of techniques. One of the most common methods is the use of ***callbacks*.**

# How Do Promises Work?

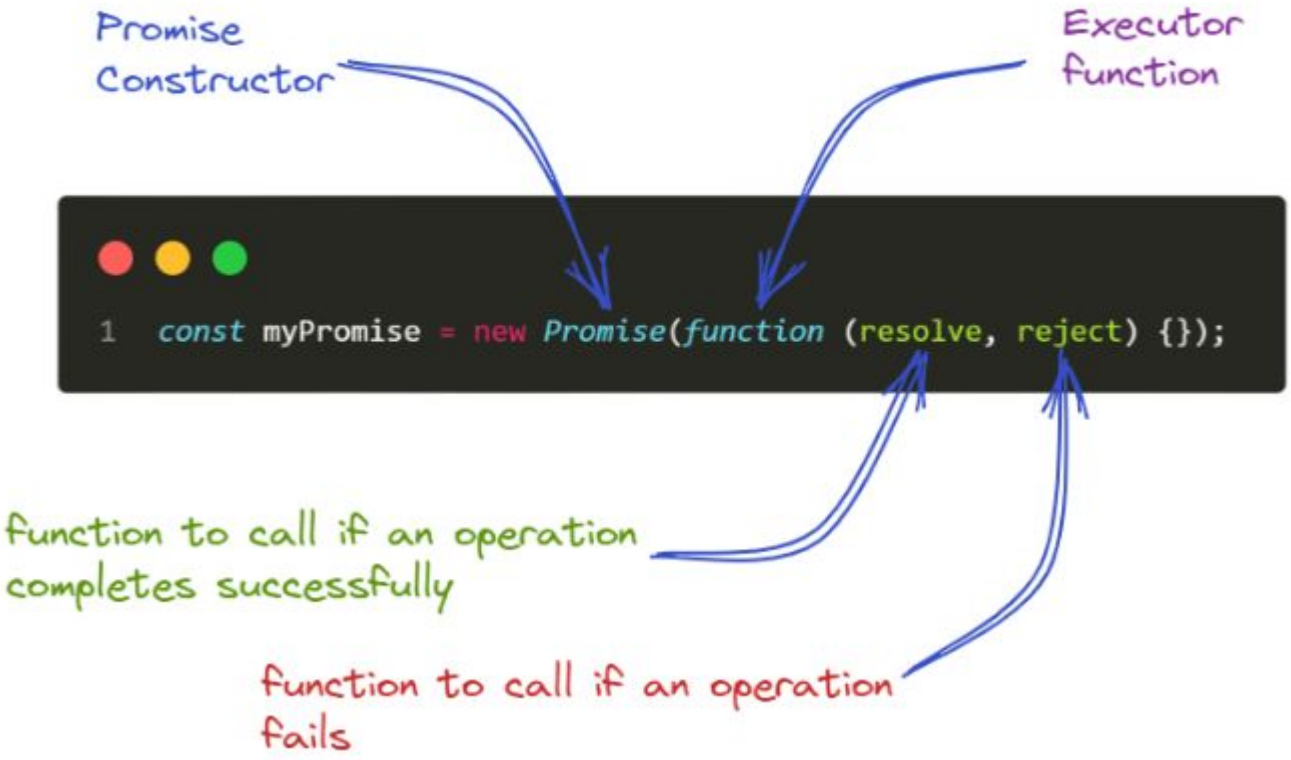
In JavaScript, a Promise is an object that will produce a single value some time in the future. If the promise is successful, it will produce a resolved value, but if something goes wrong then it will produce a reason why the promise failed. The possible outcomes here are similar to that of promises in real life.

JavaScript promises can be in one of three possible states. These states indicate the progress of the promise. They are:

- *pending*: This is the default state of a defined promise
- *fulfilled*:  This is the state of a successful promise
- *rejected*: This is the state of a failed promise



# How to Create a Promise



To create a promise, you'll create a new instance of the `Promise` object by calling the `Promise` constructor.

The constructor takes a single argument: a function called `executor`. The "executor" function is called immediately when the promise is created, and it takes two arguments: a `resolve` function and a `reject` function.

Once the promise is fulfilled, the `.then` callback method will be called with the resolved value. And if the promise is rejected, the `.catch` method will be called with an error message.

You can also add the `.finally()` method, which will be called after a promise is settled. This means that `.finally()` will be invoked regardless of the status of a promise (whether resolved or rejected).

```
myPromise
  .then((result) => {
    console.log(result);
  })
  .catch((error) => {
    console.log(error);
  })
  .finally(() => {
    //code here will be executed regardless of the status
    //of a promise (fulfilled or rejected)
  });
```

# What is Recursion in javascript

A recursive function is a function that calls itself somewhere within the body of the function. Below is a basic example of a recursive function.

As you can see, the `recursiveFunc` function calls itself within the body of the function. It will repeat calling itself until the desired output is achieved.

So how do you tell the function when to stop calling itself? You do that using a **base condition**.

```
function recursiveFunc() {
  // some code here...
  recursiveFunc()
}
```

# The Three Parts of a Recursive Function

Every time you write a recursive function, three elements must be present. They are:

- The function definition.
- The base condition.
- The recursive call.
- When these three elements are missing, your recursive function won't work as you expect. Let's take a closer look at each one.

# Why do you need a base condition?

Without the base condition, you will run into infinite recursion. A situation where your function continues calling itself without stopping, like this:

Also, without a base condition, your function exceeds the maximum call stack. You will run into the error shown below.

```
function doSomething(action) {
  console.log(`I am ${action}.`)
  doSomething(action)
}

doSomething("running")
```

```
I am running.
I am running.
I am running.
I am running.
10241 I am running.
▸Uncaught RangeError: Maximum call stack size exceeded
```

# Example of recursive function

The base condition for the `doSomething` function is `n === 0`. Anytime the function is called, it first checks if the base condition is met.

If yes, it prints `TASK COMPLETED!`. If not, it continues with the rest of the code in the function. In this case, it will print `I'm doing something.` and then call the function again.

```
function doSomething(n) {
  if(n === 0) {
    console.log("TASK COMPLETED!")
    return
  }
  console.log("I'm doing something.")
  doSomething(n - 1)
}
doSomething(3)
```

```
doSomething(3)
I'm doing something.
I'm doing something.
I'm doing something.
TASK COMPLETED!
```

**factorial with a loop:**

```
function findFactorial(num) {
  let factorial = 1
  for (let i = num; i > 0; i--) {
    factorial *= i
  }
  return factorial
}

findFactorial(5) // 120
```

**factorial with recursion:**

```
function findFactorial(num) {
  if (num === 0) return 1
  let factorial = num * findFactorial(num - 1)
  return factorial;
}

findFactorial(5) // 120
```

# Thank you