PROGRAMME ANNUEL

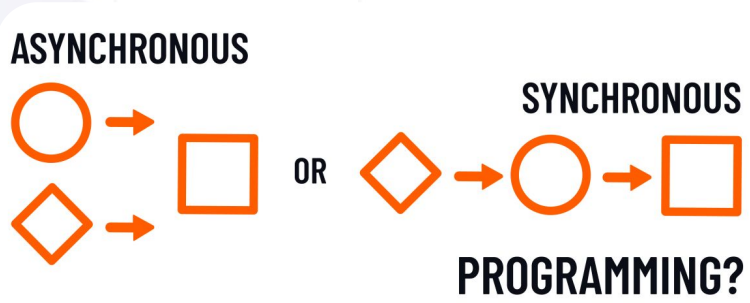
PREPARATION PFE, FULLSTACK JS

2025-2026

Sync vs Async

In computer programming and development, synchronous and asynchronous practices are essential–as long as you know the right model to use.

Synchronous tasks happen in order – you must finish the first job before moving on to the next. On the flip side, you can execute **asynchronous** jobs in any order or even simultaneously.



Sync vs Async

Synchronous , sometimes called “**sync**,” and **asynchronous** , also known as “**async**,” are two different programming models.

Understanding how these two models differ is critical for:

- Building application programming interfaces (APIs)
- Creating event-based architectures
- Deciding how to handle long-running tasks

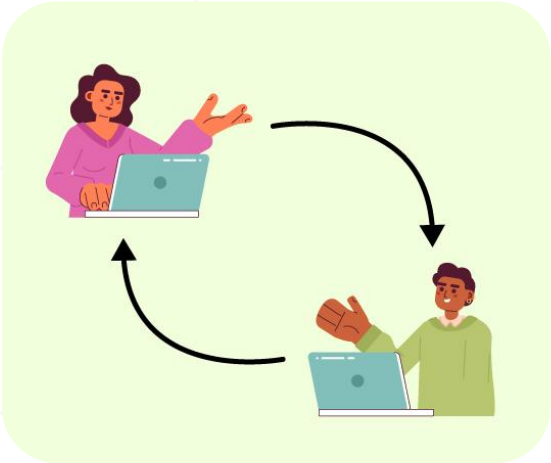
But before deciding which method to use and when, it’s important to know a few quick facts about synchronous and asynchronous programming

Synchronous

Synchronous is a blocking architecture and is best for programming reactive systems.

While one operation is being performed, other operations instructions are blocked. The completion of the first task triggers the next, and so on.

To illustrate how synchronous programming works, think of a telephone conversation. While one person speaks, the other listens. When the first person finishes, the second tends to respond immediately.



Synchronous

Tasks are executed **sequentially** , one after another.

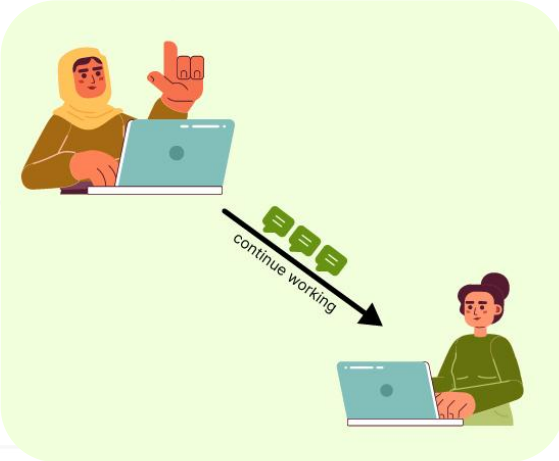
Each task must complete before the next one starts.

If one task takes a long time, it blocks the execution of subsequent tasks.

```
console.log("Task 1");
console.log("Task 2");
console.log("Task 3");
```

Asynchronous

Asynchronous is a non-blocking architecture , which means it doesn't block further execution while one or more operations are in progress. With async programming, multiple related operations can run concurrently without waiting for other tasks to complete.



OPUS LAB

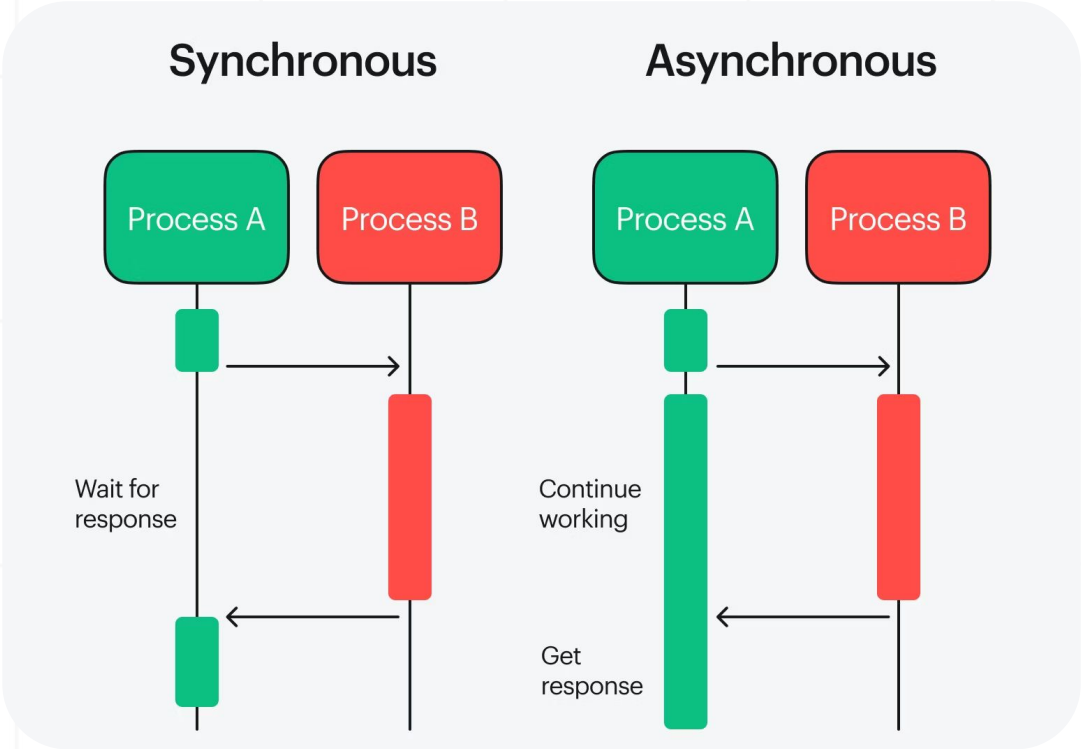
Asynchronous

- Tasks are executed independently of the main program flow.
- The program can continue executing other tasks while waiting for an asynchronous task to complete.

  Typically, uses callbacks, promises, or **async/await** .

```
console.log("Task 1");
setTimeout(() ⇒ console.log("Task 2"), 1000); // Asynchronous task
console.log("Task 3");
```

Sync vs Async



Sync vs Async

Asynchronous Programming

- Ideal for non-blocking tasks like I/O operations (e.g., API requests).
- Increases efficiency when multiple tasks can run in parallel.
- Improves application responsiveness, especially for user interfaces.

Synchronous Programming

- Best suited for operations that must occur in a strict sequence.
- Simpler to implement when tasks are interdependent.
- Provides predictability, as each task completes before the next begins.

Functions are First-Class Objects

JavaScript Functions are first-class objects in and as such can functions have the ability to:

- Be assigned to variables (and treated as a value)
- Have other functions in them
- Return other functions to be called later

Callback Functions

When a function simply accepts another function as an argument, this contained function is known as a callback function.

Using callback functions is a core functional programming concept, and you can find them in most JavaScript code; either in simple functions like **setInterval** , **setTimeout** event listening or when making API calls.

Callback Functions

Use case:

- You pass a function (goodbye) to another function (greet).
- Useful when you want something to happen **after** another task.

```
function greet(name, callback) {
 console.log("Hello " + name);
 callback(); // calling the callback function
}
function goodbye() {
 console.log("Goodbye!");
}
greet("Ahmed", goodbye);
```

What is Closure?

A closure is an **inner function** that has access to the **outer (enclosing) function's** variables – scope chain.

The closure has three scope chains:

- It has access to its own scope (variables defined between its curly brackets),
- It has access to the outer function’s variables,
- And it has access to the global variables.

What is Closure?

Callbacks are also **closures,** as the passed function is executed inside the other function just as if the callback were defined in the containing function.

Closures have access to the containing **function's scope** , so the callback function can access the containing functions' variables, and even the variables from the global scope.

What is Closure?

```
function showName (firstName, lastName) {
        var nameIntro = "Your name is ";
        // this inner function has access to the
        //outer function's variables, including the parameter

        function makeFullName () {
                return nameIntro + firstName + " " + lastName;
        }

        return makeFullName ();
}

showName ("Ahmed", "Gafsi"); // Your name is Ahmed Gafsi
```

Callback Hell

Multiple functions can be created independently and used as callback functions. These create multi-level functions. When this function tree created becomes too large, the code becomes incomprehensible sometimes and is not easily refactored. This is known as **callback hell.** Let's see an example:

Callback Hell

```
// a bunch of functions are defined up here
choosePizza(() => {
  addToppings(() => {
    bakePizza(() => {
      packPizza(() => {
        deliverPizza(() => {
          console.log("Pizza delivered! 🍕");
        });
      });
    });
  });
});
```

Callback Hell

```
if places.count > 0 {

    for i in 0..<places.count {
        for j in 0..<places.count {
            if let nameI = places[i]["name"] {
                if let cityI = places[i]["city"] {
                    if let nameJ = places[j]["name"] {
                        if let cityJ = places[j]["city"] {
                            if let latI = places[i]["lat"] {
                                if let lonI =  places[i]["lon"] {
                                    if let latitudeI = Double(latI) {
                                        if let longitudeI = Double(lonI) {
                                            if let latJ = places[j]["lat"] {
                                                if let lonJ =  places[j]["lon"] {
                                                    if let latitudeJ = Double(latJ) {
                                                        if let longitudeJ = Double(lonJ) {

                                                            if(i != j) {

                                                                let coordinateI = CLLocation(latitude: latitudeI, longitude: longitudeI)
                                                                let coordinateJ = CLLocation(latitude: latitudeJ, longitude: longitudeJ)

                                                                let distanceInMeters = coordinateI.distance(from: coordinateJ) // result is in meters
                                                                let distanceInMiles = distanceInMeters/1609.344


                                                                var distances = [Distance]()
                                                                distances.append(Distance(
                                                                    distanceInMiles: distanceInMiles,
                                                                    distanceInMeters: distanceInMeters,
                                                                    places: [
                                                                        Place(name: nameI, city: cityI, lat: latitudeI, long: longitudeI, coordinate: coordinateI),
                                                                        Place(name: nameJ, city: cityJ, lat: latitudeJ, long: longitudeJ, coordinate: coordinateJ),
                                                                        ]
                                                                ))
                                                            }
                                                        }
                                                    }
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Promises

Promises were introduced to simplify deferred activities.

Promises

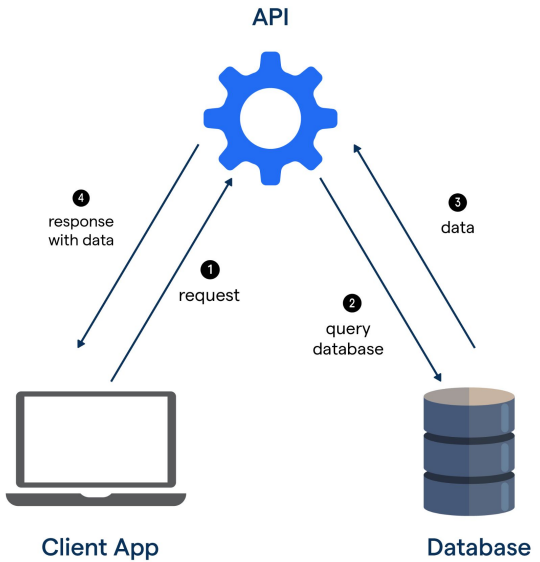
A **promise** is used to handle the **asynchronous** result of an operation.

JavaScript is designed to not wait for an **asynchronous** block of code to completely execute before other synchronous parts of the code can run.

Promises

For instance, when making API requests to servers, we have no idea if these servers are offline or online, or how long it takes to process the server request.

With Promises, we can defer execution of a code block until an async request is completed. This way, other operations can keep running without interruption.



Promises

A **Promise** is an object representing the eventual completion or failure of an asynchronous operation.

Think of it as a placeholder for a value that will be available in the future.

```
const promise = new Promise((resolve, reject) => {
  const success = true; // Simulate success or failure
  if (success) {
    resolve("Operation was successful!");
  } else {
    reject("Operation failed.");
  }
});

promise
  .then((message) => {
    console.log(message); // Output: Operation was successful!
  })
  .catch((error) => {
    console.error(error);
  });
```

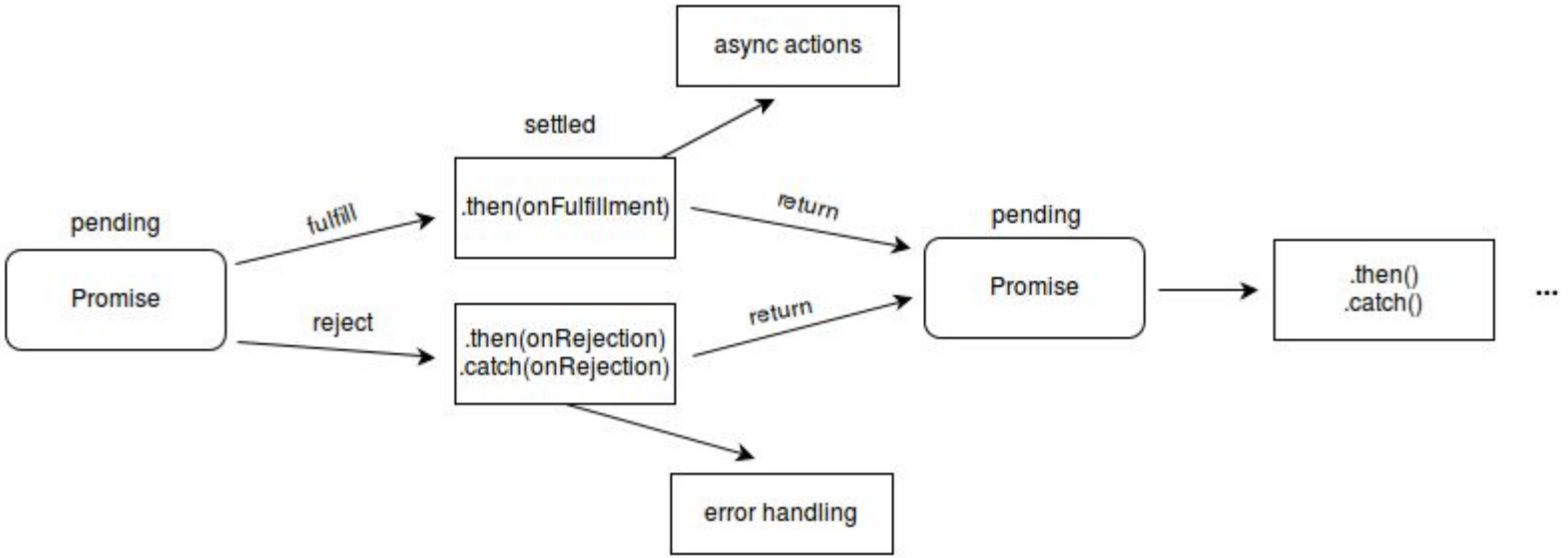
Promises

PENDING : Promise is doing work, neither then() nor catch() executes at this moment

RESOLVED : Promise is resolved ⇒ then() executes

REJECTED : Promise was rejected ⇒ catch() executes

Promises


async actions
settled
pending
fulfill
.then(onFulfillment)
return
pending
Promise
.then()
.catch()
...
Promise
reject
.then(onRejection)
.catch(onRejection)
return
error handling

Promises

When you have another **then()** block after a **catch()** or **then()** block, the promise re-enters **PENDING** mode.

(keep in mind: then() and catch() always return a new promise - either not resolving to anything or resolving to what you return inside of then()).

Only if there are no more **then()** blocks left, it enters a new, final mode: **SETTLED** .

Once **SETTLED** , you can use a special block - **finally()** - to do final cleanup work. finally() is reached no matter if you resolved or rejected before.

Promises

```
somePromiseCreatingCode()
.then(firstResult ⇒ {
return 'done with first promise';
})
.catch(err ⇒ {
// would handle any errors thrown before
// implicitly returns a new promise - just like then()
})
.finally(() ⇒ {
// the promise is settled now - finally() will NOT return a new promise!
// you can do final cleanup work here
});
```

Async / Await

Async/Await is syntactic sugar built on top of Promises. It allows you to write asynchronous code in a synchronous manner, making it easier to read and understand.

```
function fetchData() {
  return new Promise((resolve) ⇒ {
    setTimeout(() ⇒ {
      resolve("Data fetched");
    }, 2000);
  });
}

async function getData() {
  console.log("Fetching data ... ");
  const data = await fetchData();
  console.log(data); // Output: Data fetched
}

getData();
```

Example

Imagine you are a chef in a busy restaurant kitchen. You have several tasks to manage, and you need to ensure everything runs smoothly

You promise a customer that their meal will be ready in 20 minutes. This promise can either be fulfilled (meal is ready) or rejected (something went wrong).

You handle the promise like this:

Solution

```
const mealPromise = new Promise((resolve, reject) ⇒ {
  const mealReady = true; // Simulate meal preparation
  if (mealReady) {
    resolve("Meal is ready!");
  } else {
    reject("Meal preparation failed.");
  }
});

mealPromise
  .then((message) ⇒ {
    console.log(message); // Output: Meal is ready!
  })
  .catch((error) ⇒ {
    console.error(error);
  });
```

To make your workflow **smoother** , you decide to use **async/await** . You wait for the meal to be prepared before moving on to the next task

Better looking Solution

```
function prepareMeal() {
  return new Promise((resolve) ⇒ {
    setTimeout(() ⇒ {
      resolve("Meal prepared");
    }, 2000);
  });
}

async function serveMeal() {
  console.log("Preparing meal ... ");
  const meal = await prepareMeal();
  console.log(meal); // Output: Meal prepared
}

serveMeal();
```

Summary

- **Promises:** Handle asynchronous operations with .then() and .catch().
- **Async/Await:** Simplifies working with **promises** , making asynchronous code look synchronous.

These concepts are fundamental in modern JavaScript development, especially when dealing with asynchronous operations and managing scope effectively.