Purwadhika
Digital Technology School

Full Stack AI Software Development

Advanced JavaScript Concepts and Cross-Language Comparisons

Job Connector Program

Outline

Promise, Async / Await, and Callback

Learn how to handle tasks that take time, like loading data, using callbacks, Promises, and async/await.

Error Handling

Learn how to find and fix errors in your code using try...catch and other simple methods.

JSON

Understand how to use JSON to store and share data between programs or websites.

Modules

Learn how to split your code into smaller parts using import and export to keep it organized.

Cross-Language Comparisons

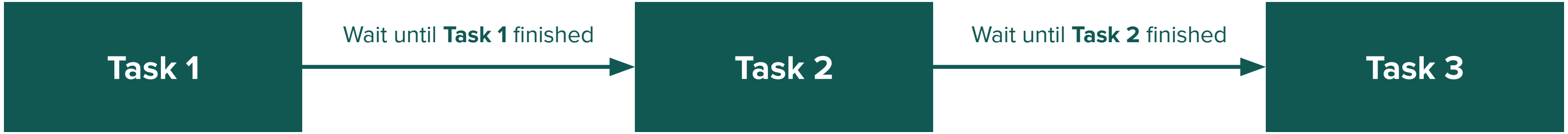
See how JavaScript features like variables, loops, and functions compare with other languages such as Java or Python.

What is Asynchronous Programming?

- JavaScript (and TypeScript) runs single-threaded.
- Long-running operations (e.g., fetching data) block execution.
- Asynchronous programming allows other tasks to run while waiting for a process to complete.

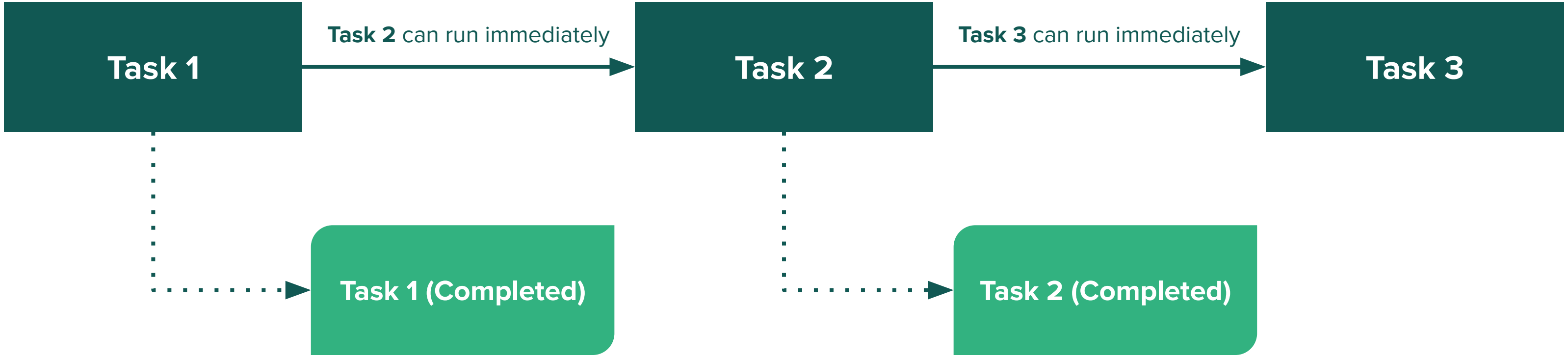
Asynchronous VS Synchronous

Synchronous flow in JavaScript means each task runs one after another. **The next task starts only after the previous one is finished.**


Task 1
Wait until **Task 1** finished
Task 2
Wait until **Task 2** finished
Task 3

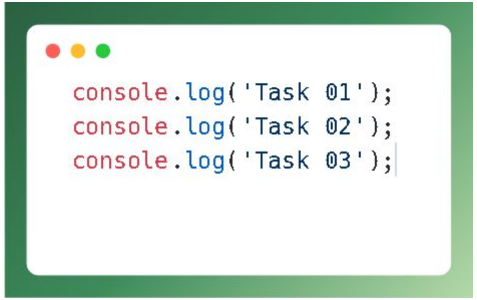
Asynchronous VS Synchronous

Asynchronous flow in JavaScript means tasks can run independently. **The next task can start even if the previous one isn't finished yet.** This is common with **promises** or **callbacks**, which allow JavaScript to keep running without waiting.


Task 1
Task 2 can run immediately
Task 2
Task 3 can run immediately
Task 3
Task 1 (Completed)
Task 2 (Completed)

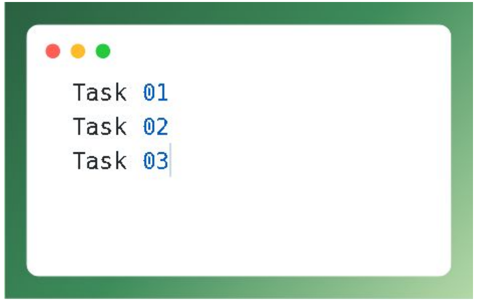
Synchronous Example Code

As we can see, the code is very simple and the flow is clear. ***console.log("Task 1")*** will run first, then ***console.log("Task 2")***, then finally ***console.log("Task 3")***, there is no asynchronous code here.



```
console.log('Task 01');
console.log('Task 02');
console.log('Task 03');
```

Expected Output

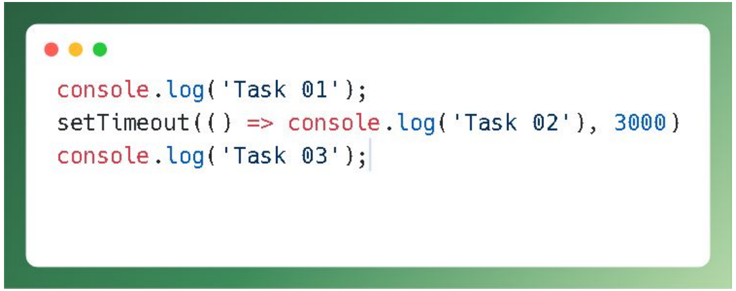


```
Task 01
Task 02
Task 03
```

Asynchronous Example Code

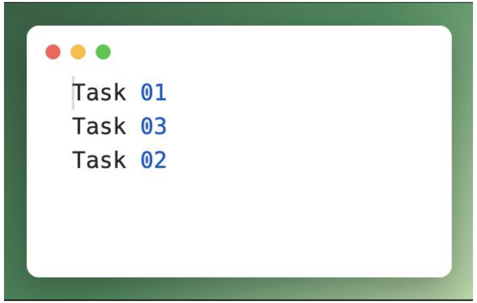
In the example, ***setTimeout()*** in Task 2 takes 3 seconds to finish. The result shows Task 1, Task 3, then Task 2.

This happens because ***setTimeout()*** runs in the background while the next tasks keep running. When it's done, Task 2's result appears.



```
console.log('Task 01');
setTimeout(() => console.log('Task 02'), 3000)
console.log('Task 03');
```

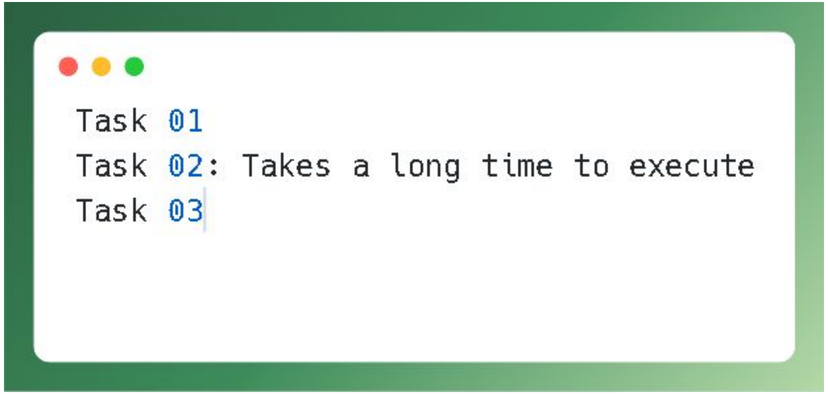
Expected Output



```
Task 01
Task 03
Task 02
```

Why Asynchronous Important?

If our code runs synchronously, it takes a long time to finish because **Task 2** blocks the next tasks while it's running. By making the code asynchronous, other tasks can keep running without waiting for **Task 2** to complete.



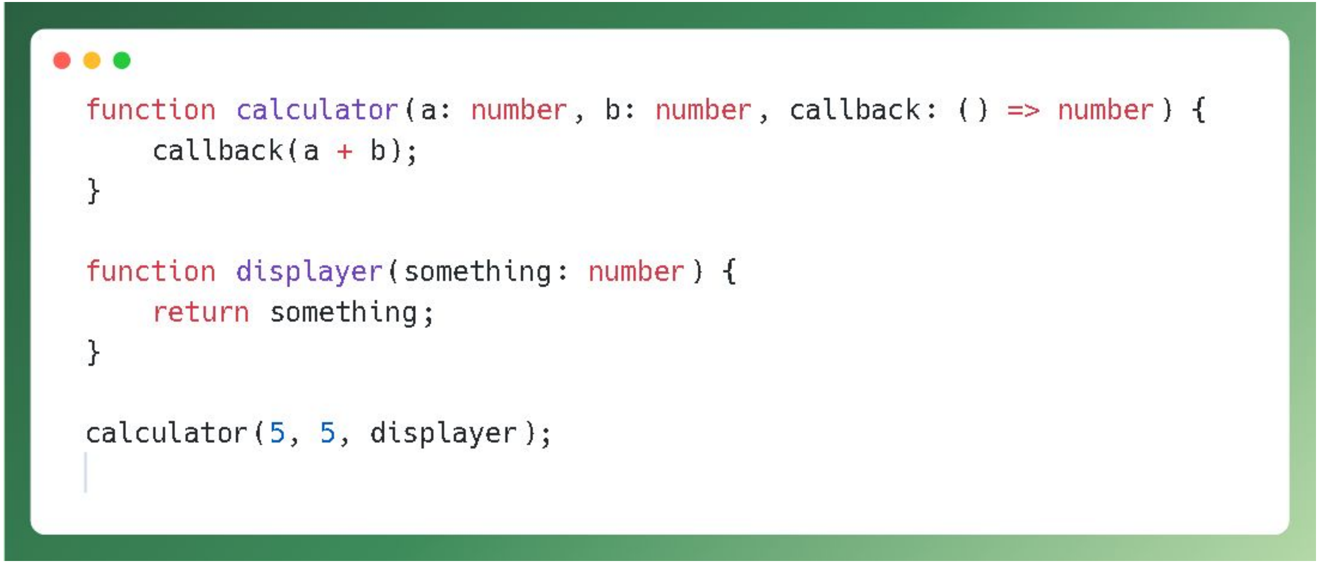
```
Task 01
Task 02: Takes a long time to execute
Task 03
```

We will explore three patterns to implement asynchronous: **Callbacks, Promises, and Async/Await.**

Callbacks

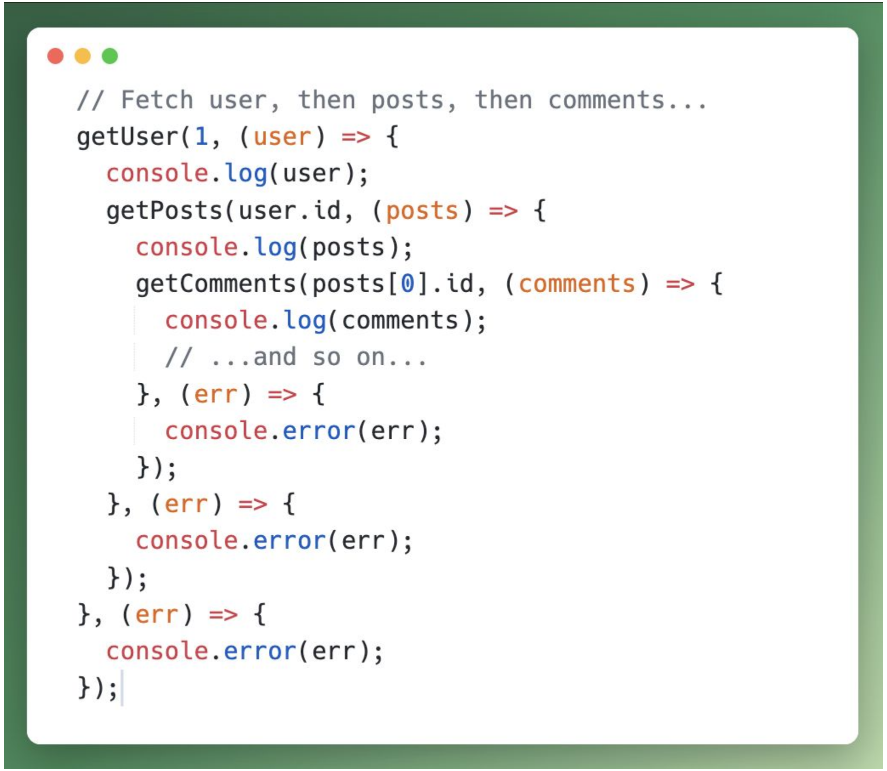
- A **callback** is a function passed as an argument to another function, which is then executed when the async operation completes.
- Concept: "Call this function back when you're done."
- Problem: "Callback Hell" or "Pyramid of Doom", deeply nested, hard-to-read, and error-prone code.

Callbacks - Example



```
function calculator(a: number, b: number, callback: () => number) {
    callback(a + b);
}

function displayer(something: number) {
    return something;
}

calculator(5, 5, displayer);
```

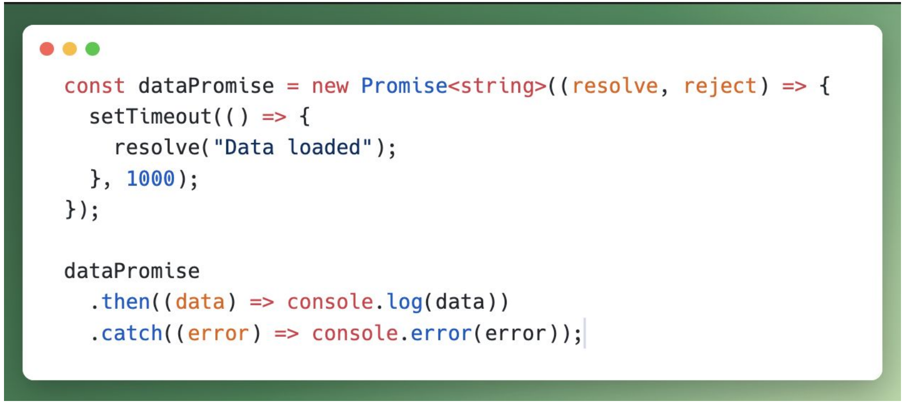
Callback Hell



```
// Fetch user, then posts, then comments...
getUser(1, (user) => {
  console.log(user);
  getPosts(user.id, (posts) => {
    console.log(posts);
    getComments(posts[0].id, (comments) => {
      console.log(comments);
      // ...and so on...
    }, (err) => {
      console.error(err);
    });
  }, (err) => {
    console.error(err);
  });
}, (err) => {
  console.error(err);
});
```

Promises

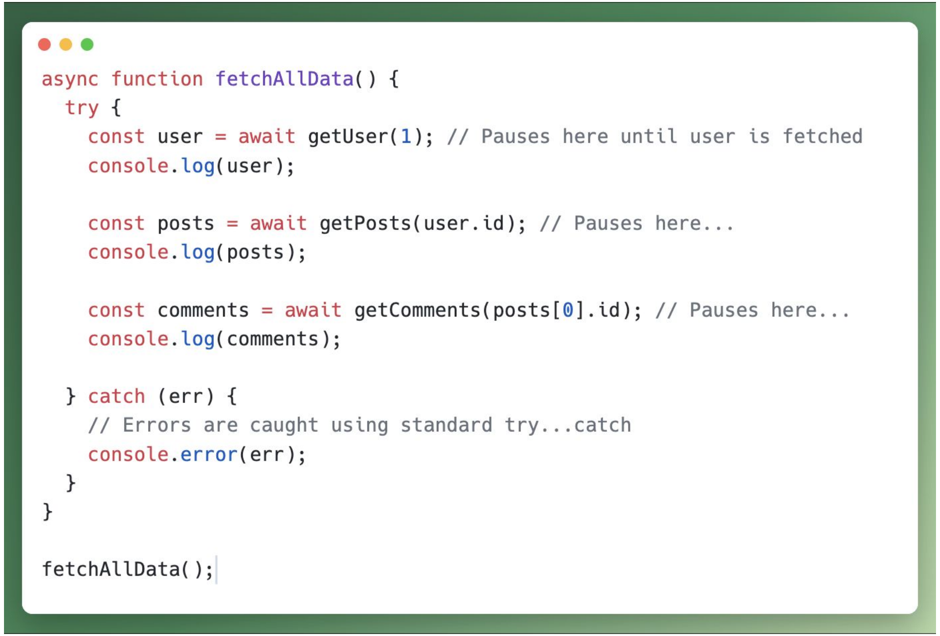
- A **Promise** is an object representing the eventual completion (or failure) of an asynchronous operation.
- States:
  - Pending: Initial state, not yet fulfilled or rejected.
  - Fulfilled (Resolved): The operation completed successfully.
  - Rejected: The operation failed.
- Chaining: Uses **.then()** for success and **.catch()** for errors, creating a much cleaner, linear flow.



```
const dataPromise = new Promise<string>((resolve, reject) => {
  setTimeout(() => {
    resolve("Data loaded");
  }, 1000);
});

dataPromise
  .then((data) => console.log(data))
  .catch((error) => console.error(error));
```

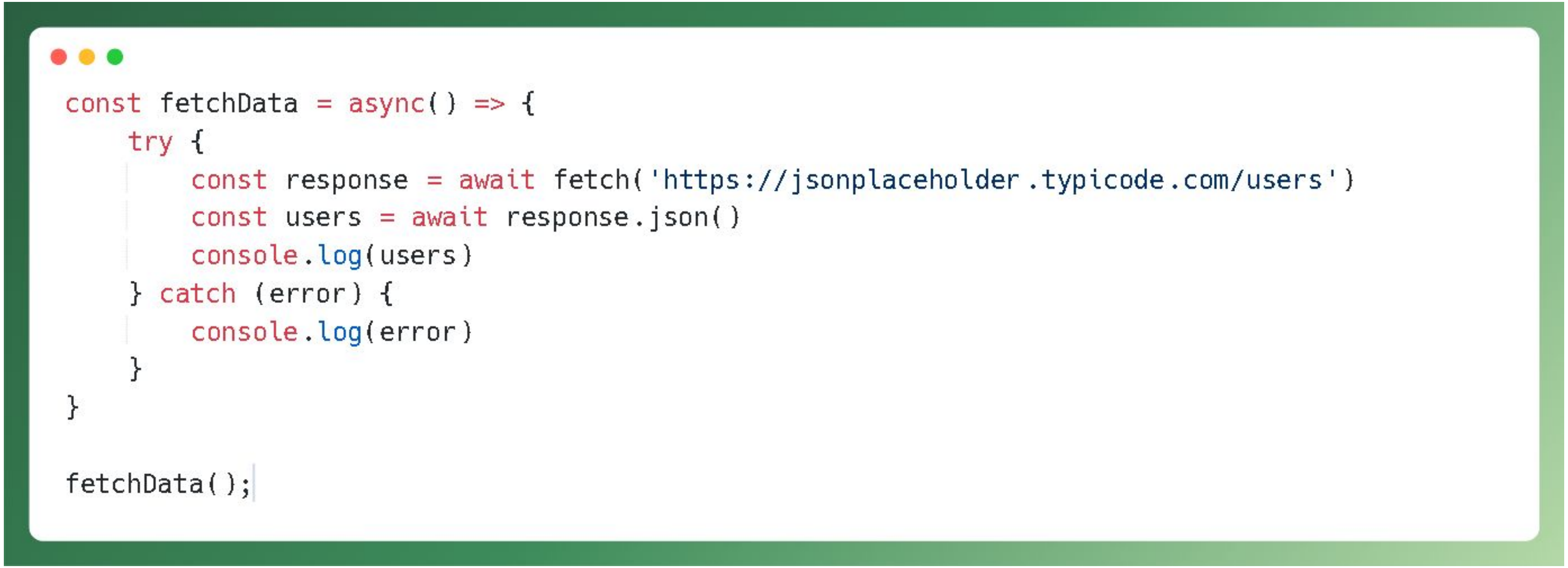
Async / Await

- Async/Await is modern "syntactic sugar" built on top of Promises. It makes your async code look and behave like synchronous code.
- **async**: A keyword placed before a function declaration. It makes the function automatically return a Promise.
- **await**: A keyword that pauses the async function execution until a Promise is settled (resolved or rejected).
  - It can only be used inside an async function.
- Benefit: Cleaner syntax and better error handling.



```
async function fetchAllData() {
  try {
    const user = await getUser(1); // Pauses here until user is fetched
    console.log(user);

    const posts = await getPosts(user.id); // Pauses here...
    console.log(posts);

    const comments = await getComments(posts[0].id); // Pauses here...
    console.log(comments);

  } catch (err) {
    // Errors are caught using standard try...catch
    console.error(err);
  }
}

fetchAllData();
```

Real Case Implementation with Async / Await

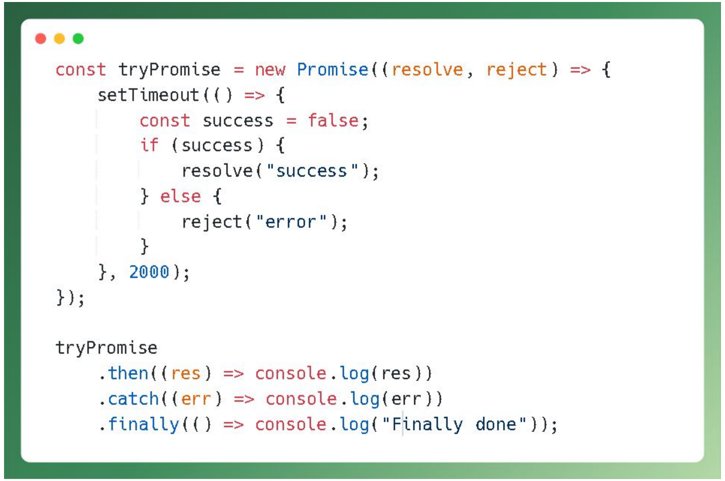


```
const fetchData = async() => {
    try {
        const response = await fetch('https://jsonplaceholder.typicode.com/users')
        const users = await response.json()
        console.log(users)
    } catch (error) {
        console.log(error)
    }
}

fetchData();
```

Error Handling

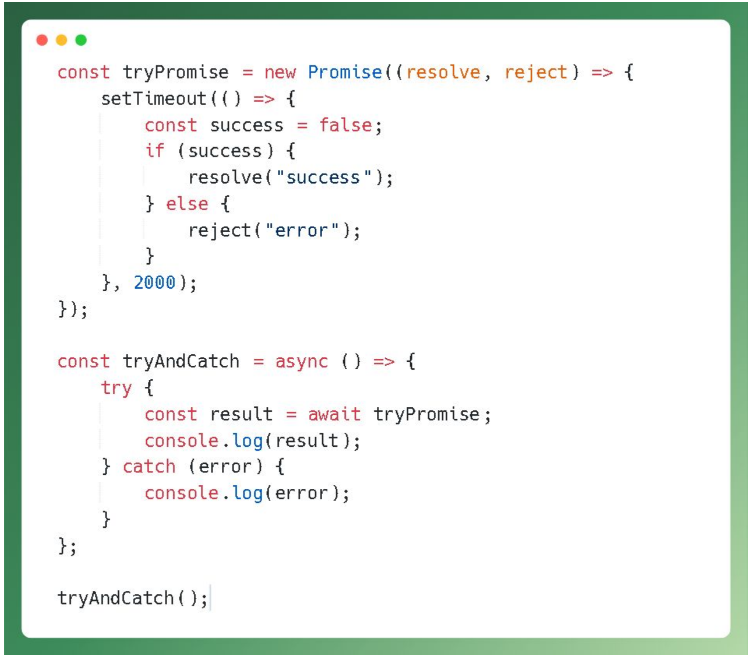
Errors are inevitable, from invalid API responses to network issues. Proper handling prevents app crashes.

Promise



```javascript
const tryPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
        const success = false;
        if (success) {
            resolve("success");
        } else {
            reject("error");
        }
    }, 2000);
});

tryPromise
    .then((res) => console.log(res))
    .catch((err) => console.log(err))
    .finally(() => console.log("Finally done"));
```

Async/Await

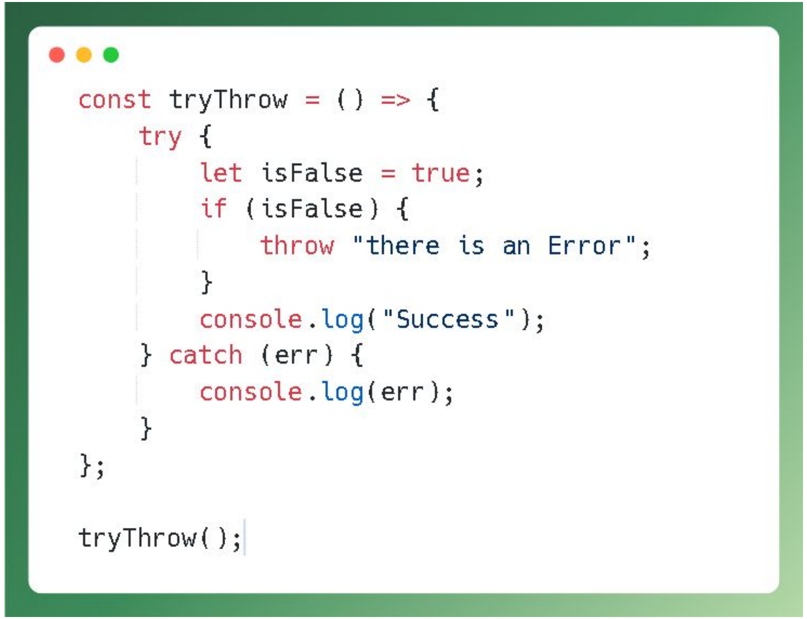


```javascript
const tryPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
        const success = false;
        if (success) {
            resolve("success");
        } else {
            reject("error");
        }
    }, 2000);
});

const tryAndCatch = async () => {
    try {
        const result = await tryPromise;
        console.log(result);
    } catch (error) {
        console.log(error);
    }
};

tryAndCatch();
```

Error Handling: Throw

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**.

Usually throw statement used together with try and catch, you can control program flow and generate custom error messages.

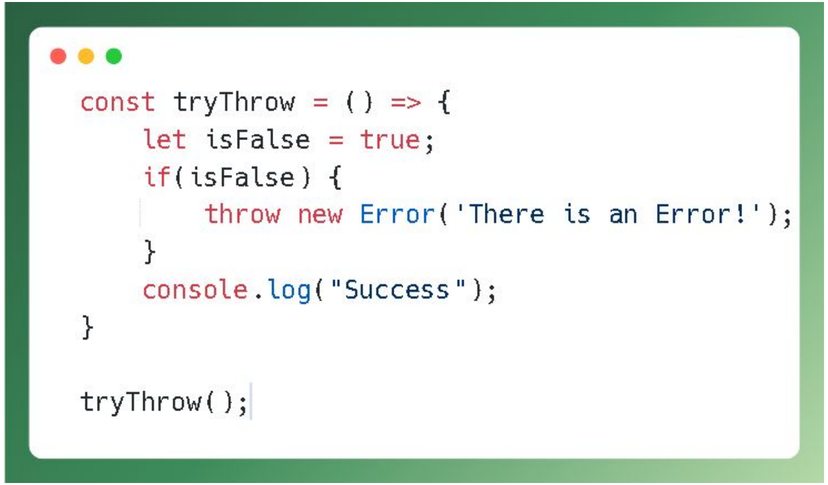


```
const tryThrow = () => {
    try {
        let isFalse = true;
        if (isFalse) {
            throw "there is an Error";
        }
        console.log("Success");
    } catch (err) {
        console.log(err);
    }
};

tryThrow();
```

Error Handling: Throw with Error()

We can also use throw with **new Error()**.

When **isFalse** = **true**, then throw **new Error("There is an Error!")**. That function will be stop and console.log(“Success) will not execute and will return an **Error: "There is an Error!"**.



```
const tryThrow = () => {
    let isFalse = true;
    if(isFalse) {
        throw new Error('There is an Error!');
    }
    console.log("Success");
}

tryThrow();
```

JSON (JavaScript Object Notation)

What is JSON?

- Lightweight data-interchange format.
- Text-based representation of objects and arrays.
- Used across APIs and configurations.

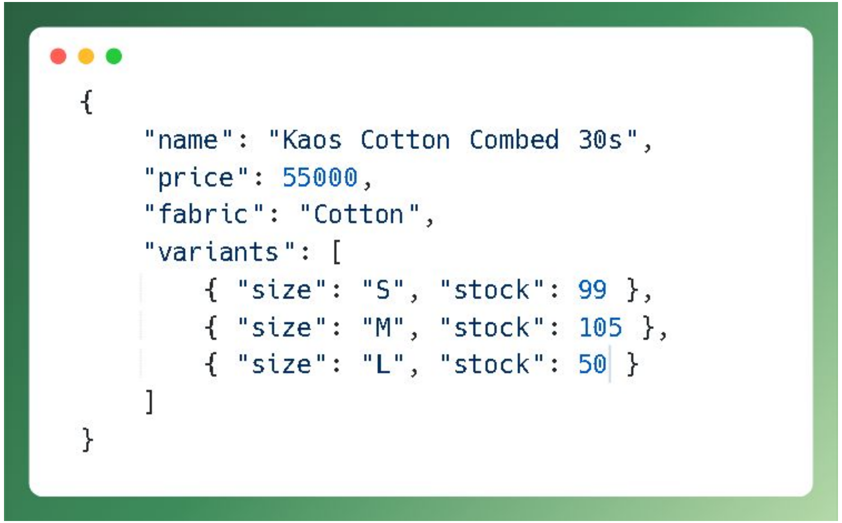
JSON (JavaScript Object Notation)

JSON syntax rules:

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

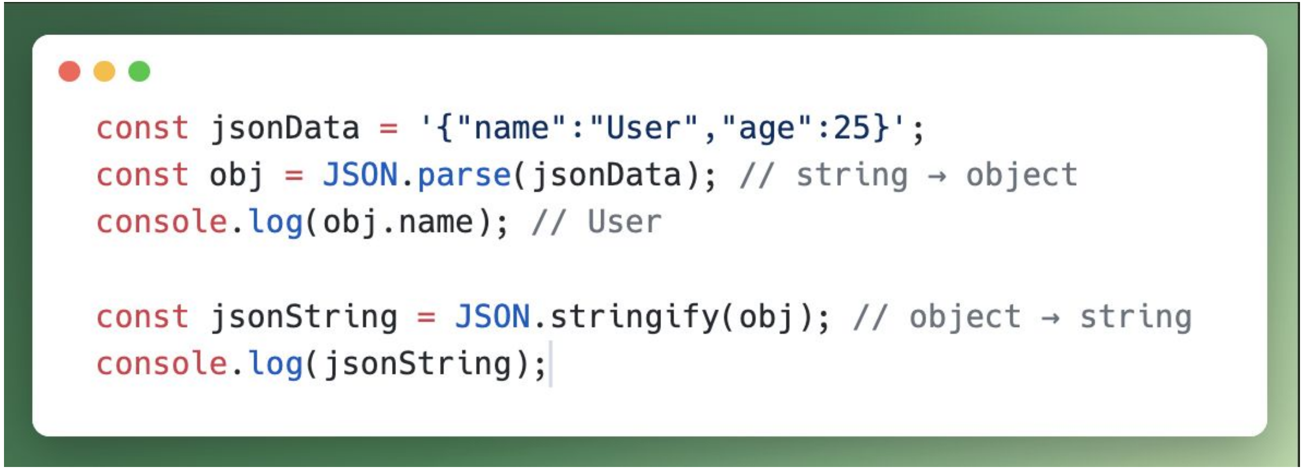
The JSON format is syntactically identical to the code for creating JavaScript objects.

Because of this similarity, a JavaScript program can easily convert JSON data into native JavaScript objects.



```
{
    "name": "Kaos Cotton Combed 30s",
    "price": 55000,
    "fabric": "Cotton",
    "variants": [
        { "size": "S", "stock": 99 },
        { "size": "M", "stock": 105 },
        { "size": "L", "stock": 50 }
    ]
}
```

Using JSON in TypeScript



```
const jsonData = '{"name":"User","age":25}';
const obj = JSON.parse(jsonData); // string → object
console.log(obj.name); // User

const jsonString = JSON.stringify(obj); // object → string
console.log(jsonString);
```

TypeScript Modules

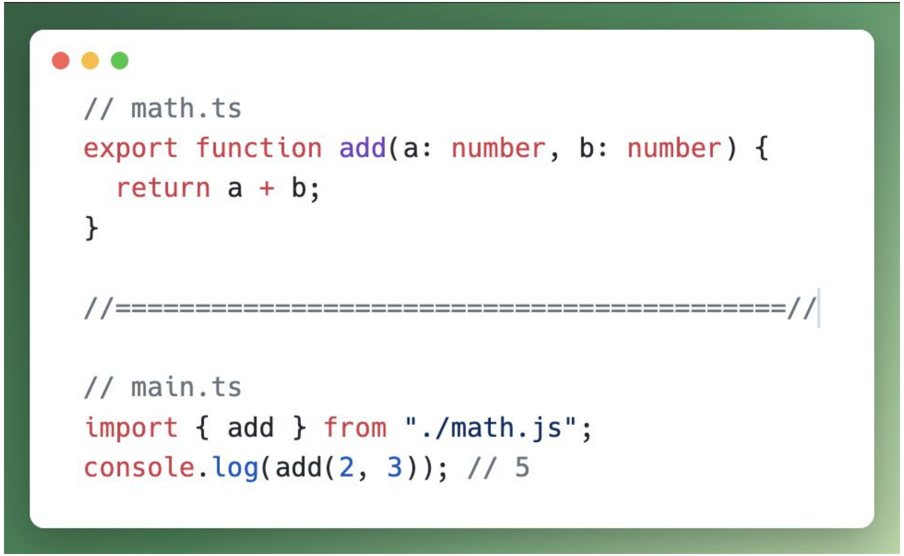
What are Modules? Modules allow you to split code into reusable files.

Benefits:

- Code reusability
- Better organization
- Scope isolation (no global pollution)

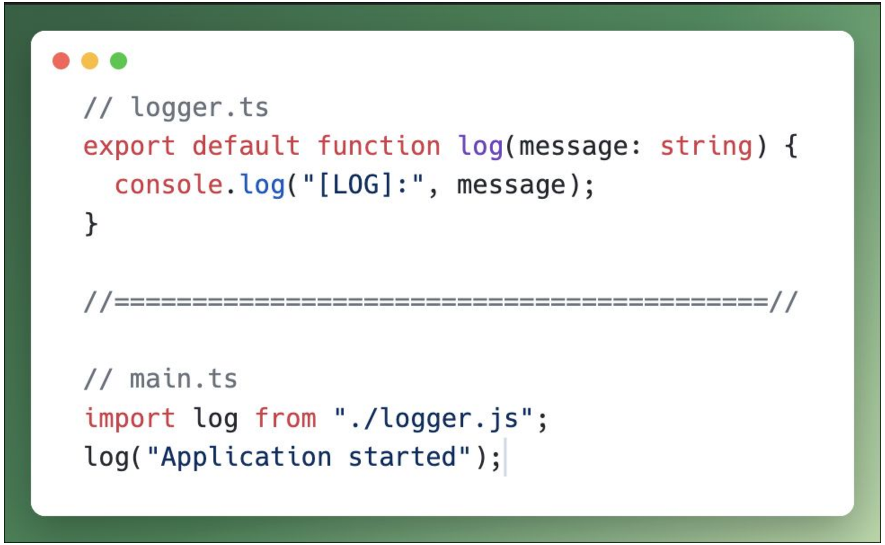
Typescript Modules

Named Export:



```
// math.ts
export function add(a: number, b: number) {
  return a + b;
}

//==========================================//

// main.ts
import { add } from "./math.js";
console.log(add(2, 3)); // 5
```

Default Export:



```
// logger.ts
export default function log(message: string) {
  console.log("[LOG]:", message);
}

//==========================================//

// main.ts
import log from "./logger.js";
log("Application started");
```

Cross-Language Comparisons

What is TypeScript?
Before we compare, let's establish our baseline. TypeScript is not a new, independent language.

- **It’s a Superset of JavaScript**
  Any valid JavaScript is valid TypeScript. It transpiles (compiles) down to plain JavaScript.
- **It’s Core Feature is Static Typing**
  It adds a powerful type system on top of JavaScript's dynamic nature.
- **It’s Goal is Scale and Safety**
  It was designed by Microsoft to help build and maintain massive, complex JavaScript applications by catching errors at compile-time (before they reach the user).
- **It’s Typing is Structural**
  This is a critical concept. A type is defined by its shape (what properties and methods it has), not by its name. This is also called "duck typing, but at compile-time."

The Contenders

A brief overview of the languages we're comparing TypeScript (as a backend platform via Node.js) against.

Python

The high-level, dynamically typed "batteries included" language. Dominant in **AI, data science, and scripting**.

Go (Golang)

The minimalist, statically typed, compiled language from Google. Built for **concurrency, cloud infrastructure, and CLI tools**.

JAVA

The enterprise-standard, statically typed, class-based OOP language. Runs on the JVM and powers **large-scale enterprise backends and Android.**

PHP

The server-side scripting language that powers a vast portion of the web. Dominant in **content management (e.g., WordPress) and frameworks like Laravel.**

The Core Difference: Asking for ID vs. Checking the Wallet

Language	The Typing Paradigm	The Analogy	What It Means
TypeScript (TS)	**Structural Typing**	**The Wallet Check:** "I don't care what your name is. Do you have $5 and a driver's license? Yes? You can come in."	Very **flexible** and works well with JavaScript's unpredictable nature.
Java & PHP (Modern)	**Nominal Typing**	**The ID Check:** "I don't care what's in your wallet. Is your ID card labeled Person? If not, you can't come in."	Very **strict** and safe. You always know exactly what a piece of data is meant to be.
Python	**Dynamic/Gradual Typing**	**The Trial-and-Error Check:** "Just come in. We'll see if you crash into anything. (Optional: You can leave a sticky note on your shirt saying what you *should* be.)"	**Fastest to start**, but you find mistakes only when the program is running.
Go (Golang)	**Hybrid**	**The Smart Check:** "I need you to open this door. I don't care if you're a Guard or a Janitor—do you have a Key? (The Key is the 'interface')."	**A good balance** of strictness and flexibility.

Handling Multiple Tasks (Concurrency)

Language	The Concurrency Model	The Kitchen Analogy 🧑‍🍳	What It's Best For
TypeScript (Node.js)	**Single-Threaded Event Loop**	**One Super-Chef:** The chef takes an order, starts the water boiling (slow I/O), immediately moves to the *next* order while the water heats up, and comes back when the timer goes off.	**I/O Heavy:** Great for web servers that mostly wait for the network or database.
Go (Golang)	**Goroutines & Channels**	**Thousands of Lightweight Chefs:** The manager instantly spawns thousands of tiny, cheap chefs that all work at the same time on different CPUs. They pass notes ("channels") to communicate.	**Speed & Scale:** Fantastic for any kind of heavy-duty, simultaneous work (CPU or I/O).
Java (Modern)	**Virtual Threads**	**Industrial Robot Chefs:** Used to be slow, heavy robots. Now, thanks to "Virtual Threads," it can instantly spawn millions of lightweight chefs too!	**Enterprise Power:** The best of the old (stability) and the new (speed).
Python	**The GIL & Async**	**One Chef With a Waiter:** The waiter (`asyncio`) is great at juggling I/O tasks, but the main chef (`GIL`) can only handle one CPU-intensive part of the recipe at a time.	**Data Science:** Useful because data libraries (like NumPy) use C code, which lets them skip the slow Python chef.
PHP	**Share-Nothing**	**Disposable Kitchens:** Every customer request gets a brand-new kitchen built just for them. When the food is served, the whole kitchen is thrown away.	**Stateless Websites:** Very robust and simple. One crash can't hurt the next customer.

Simple Syntax Comparison: Writing the Code

Feature	TypeScript (TS)	Python	Go (Golang)	Java	PHP (Modern)
New Variable	const name: string = "Alice";	name = "Alice"	name := "Alice"	String name = "Alice";	$name = "Alice";
Defining a Function	function add(a: number, b: number): number { ... }	def add(a: int, b: int) -> int: ...	func add(a, b int) int { ... }	public static int add(int a, int b) { ... }	function add(int $a, int $b): int { ... }
If Something Goes Wrong (Error)	try { } catch (e) { }	try: except Exception as e:	**if err != nil { ... }** (Unique to Go)	try { } catch (Exception e) { }	try { } catch (\Throwable $e) { }
Style/Look	Very clean, uses braces {}.	Very clean, uses **indentation** to group code.	Simple, requires very few keywords, looks minimalist.	The most "wordy" (verbose) with lots of required keywords.	Looks a lot like TS/Java with the added dollar signs $.

When to Pick Which Tool?

Language	Best Use Case	Why Choose It Over TS?
TypeScript	Building **large, complex websites** and **apps** where you need code to be safe and easily managed by a team.	If your company already uses JavaScript everywhere, TS is the natural, safest upgrade.
Go (Golang)	Building **super-fast background services** (microservices), command-line tools, or cloud infrastructure.	You need the **fastest possible performance** and the lowest memory usage for heavy workloads.
Java	Building **massive, stable enterprise applications** (like banking or insurance systems).	You need the **most mature ecosystem**, extreme stability, and powerful, long-running features.
Python	**Data Science, AI, Machine Learning,** and simple, quick web backends.	Your primary goal is using powerful **data libraries** or getting a quick prototype out the door.
PHP	Building **content-driven websites** (like blogs or e-commerce) or using powerful frameworks like Laravel.	You want a simple, **deployment-friendly** language focused purely on the web server.

Exercise

- Create a function to merge two array of student data and remove duplicate data
- Student data : name & email
- Example :

```
Array1 → [
          { name: ‘Student 1’, email : ‘student1@mail.com’  },
          { name: ‘Student 2’, email : ‘student2@mail.com’  }
]
Array2 → [
          { name: ‘Student 1’, email : ‘student1@mail.com’  },
          { name: ‘Student 3’, email : ‘student3@mail.com’  }
]
```

- Result :

```
ArrayResult → [
          { name: ‘Student 1’, email : ‘student1@mail.com’  },
          { name: ‘Student 2’, email : ‘student2@mail.com’  },
          { name: ‘Student 3’, email : ‘student3@mail.com’  }
]
```

Exercise

- Create a function that can accept input as an array of objects and switch all values into property and property into value
- Example :
    - Input : **`[{ name: ‘David’, age: 20 }]`**
    - Output : **`[{ David: ‘name’, 20: ‘age’}]`**

Exercise

- Create a function to find a factorial number using recursion
- Example
  - Input : **`5`**
  - Output: **`5! = 5 x 4 x 3 x 2 x 1 = 120`**

Thank you