ES6 Day2

By:Nawal Zaki Abdelhady

Content

Call Stack

The Call Stack is a data structure that keeps track of function calls.
JavaScript is single-threaded, so it can only execute one function at a time.
Functions are added to the stack when called, and removed when finished.

```
function first() {
  console.log("First function");
  second();
}

function second() {
  console.log("Second function");
  third();
}

function third() {
  console.log("Third function");
}

first();
```

```
First function
Second function
Third function
```

Explanation:

- `first()` is added to the **Call Stack** → calls `second()` → `third()`
- Functions execute in **LIFO order** (Last In, First Out)

Event Queue

The Event Queue (or Task Queue) stores asynchronous tasks (like timers, fetch responses, or events) waiting to be executed. Once the Call Stack is empty, the Event Loop moves tasks from the Event Queue to the Call Stack.

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

setTimeout(() => {
  console.log("Inside setTimeout");
}, 1000);

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

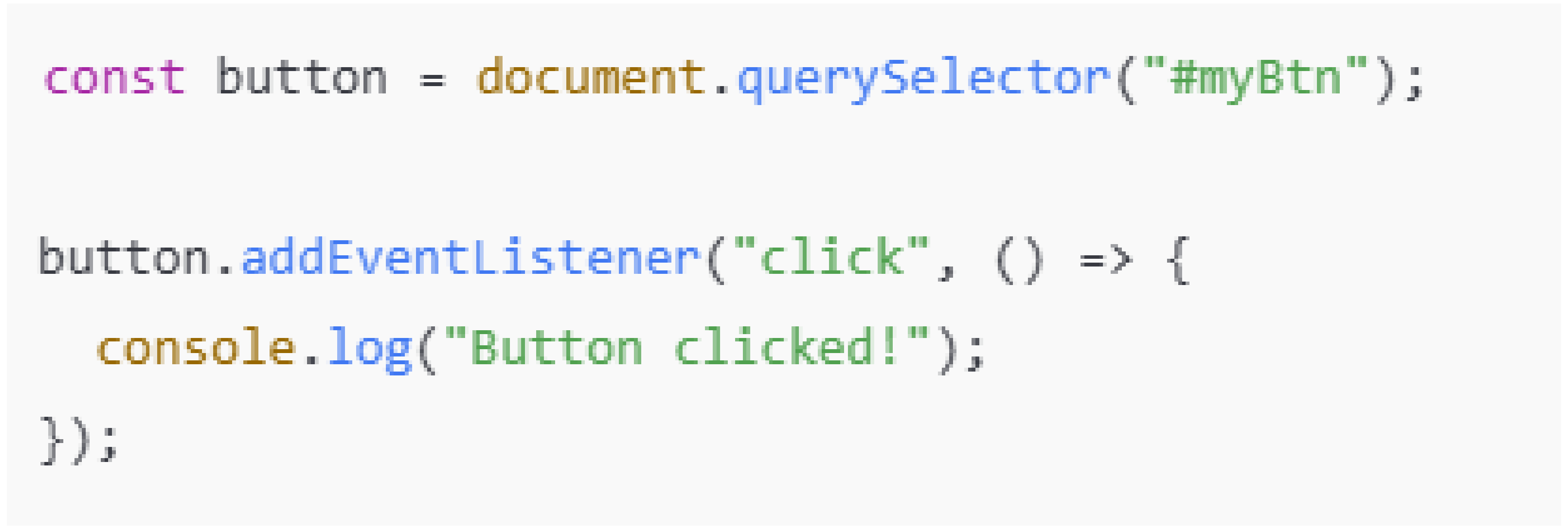
```
Start
End
Inside setTimeout
```

Explanation:

- `setTimeout` is async → its callback goes to **Event Queue**
- Call Stack executes synchronous code first (`Start` → `End`)
- After 1 second, callback executes from **Event Queue**

Events

Events are actions triggered by the user or browser (click, scroll, input, load).
Event listeners handle these events.

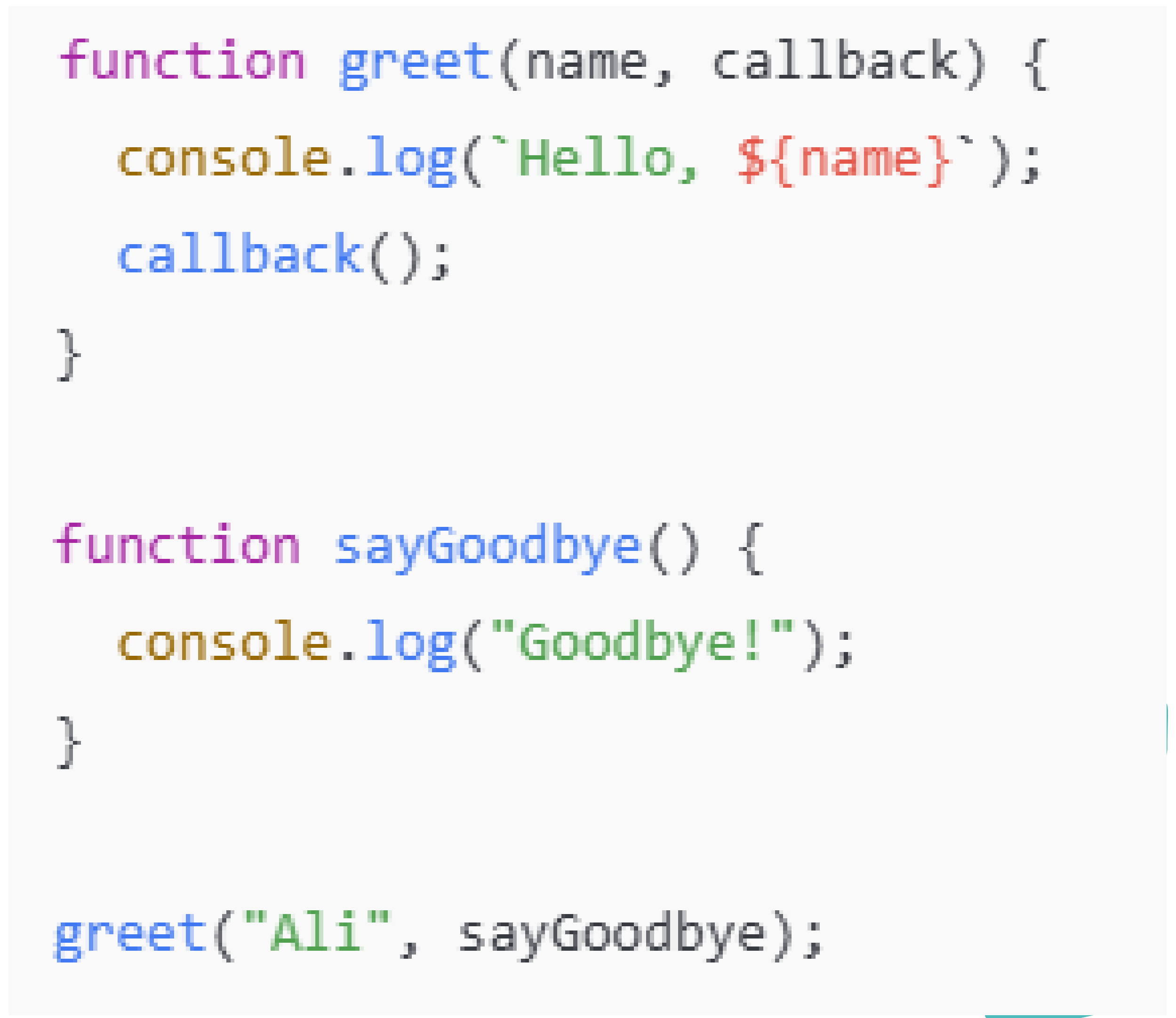


```
const button = document.querySelector("#myBtn");

button.addEventListener("click", () => {
  console.log("Button clicked!");
});
```

Callback Functions

A callback function is a function passed as an argument to another function, to be executed later.

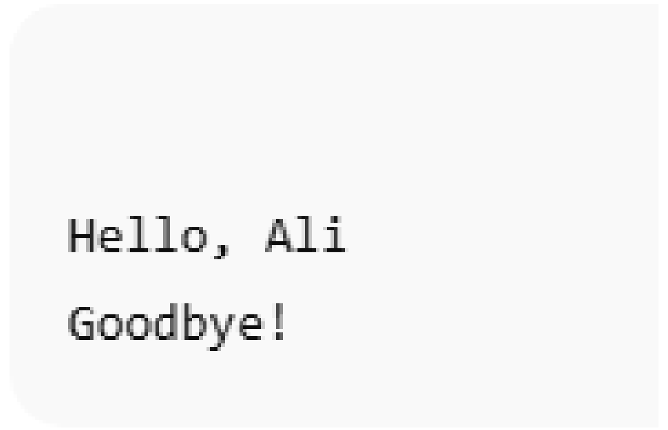


```
function greet(name, callback) {
  console.log(`Hello, ${name}`);
  callback();
}

function sayGoodbye() {
  console.log("Goodbye!");
}

greet("Ali", sayGoodbye);
```

Output:



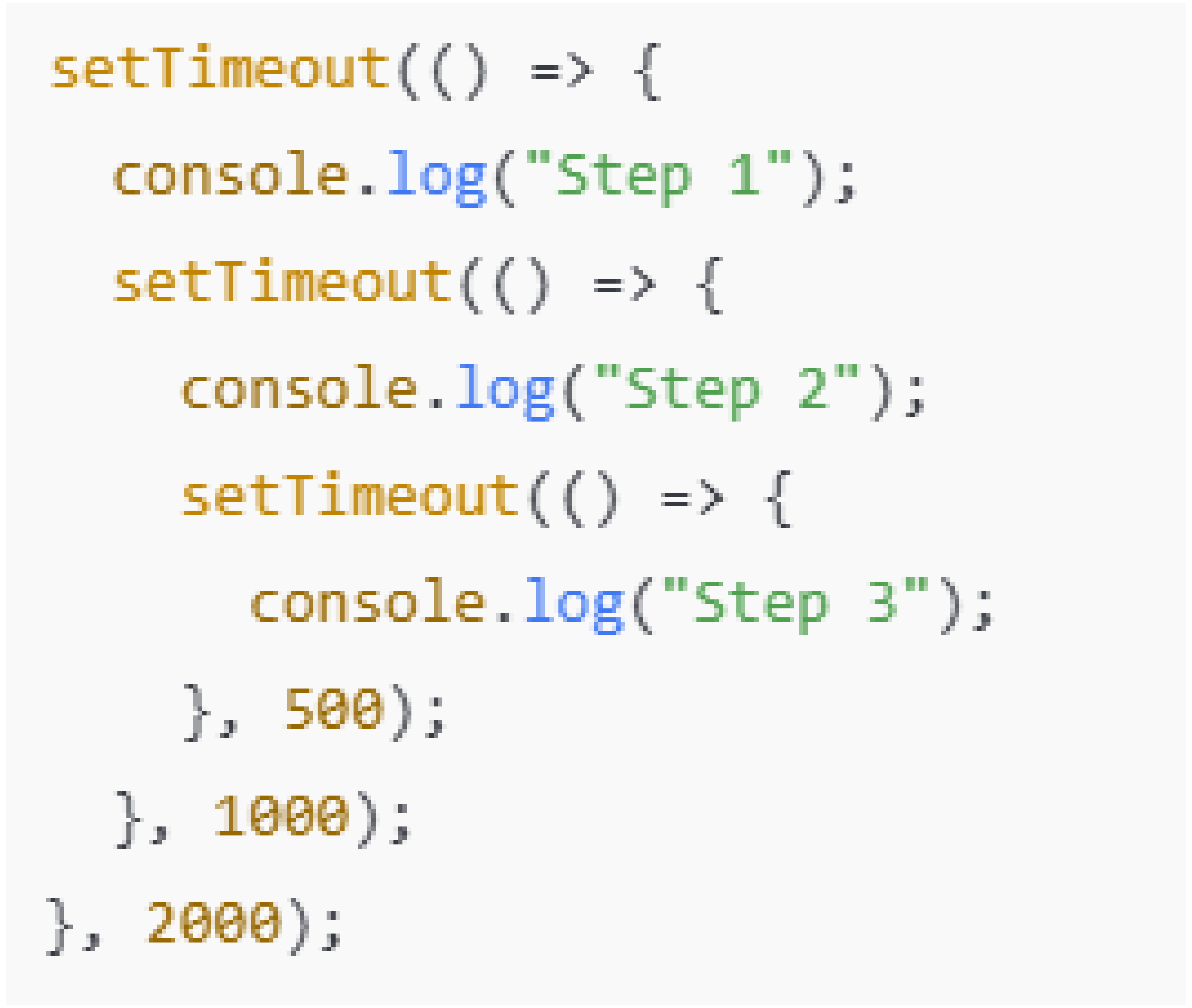
```
Hello, Ali
Goodbye!
```

Callback Hell

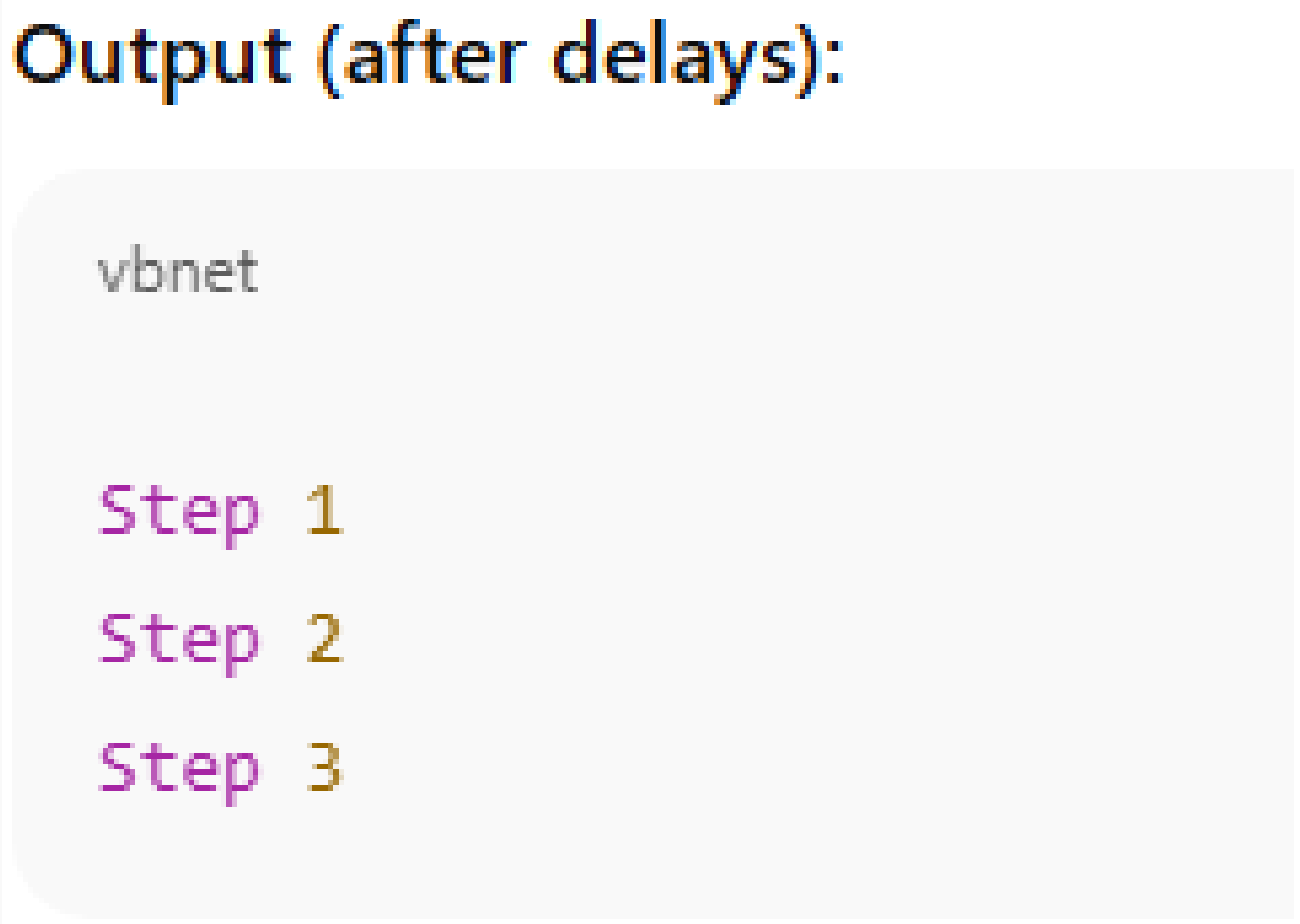
Callback Hell occurs when callbacks are nested deeply, making code hard to read and maintain.

Problem:

- Code becomes deeply nested (“pyramid of doom”)
- Hard to debug and maintain



```
setTimeout(() => {
  console.log("Step 1");
  setTimeout(() => {
    console.log("Step 2");
    setTimeout(() => {
      console.log("Step 3");
    }, 500);
  }, 1000);
}, 2000);
```



Output (after delays):

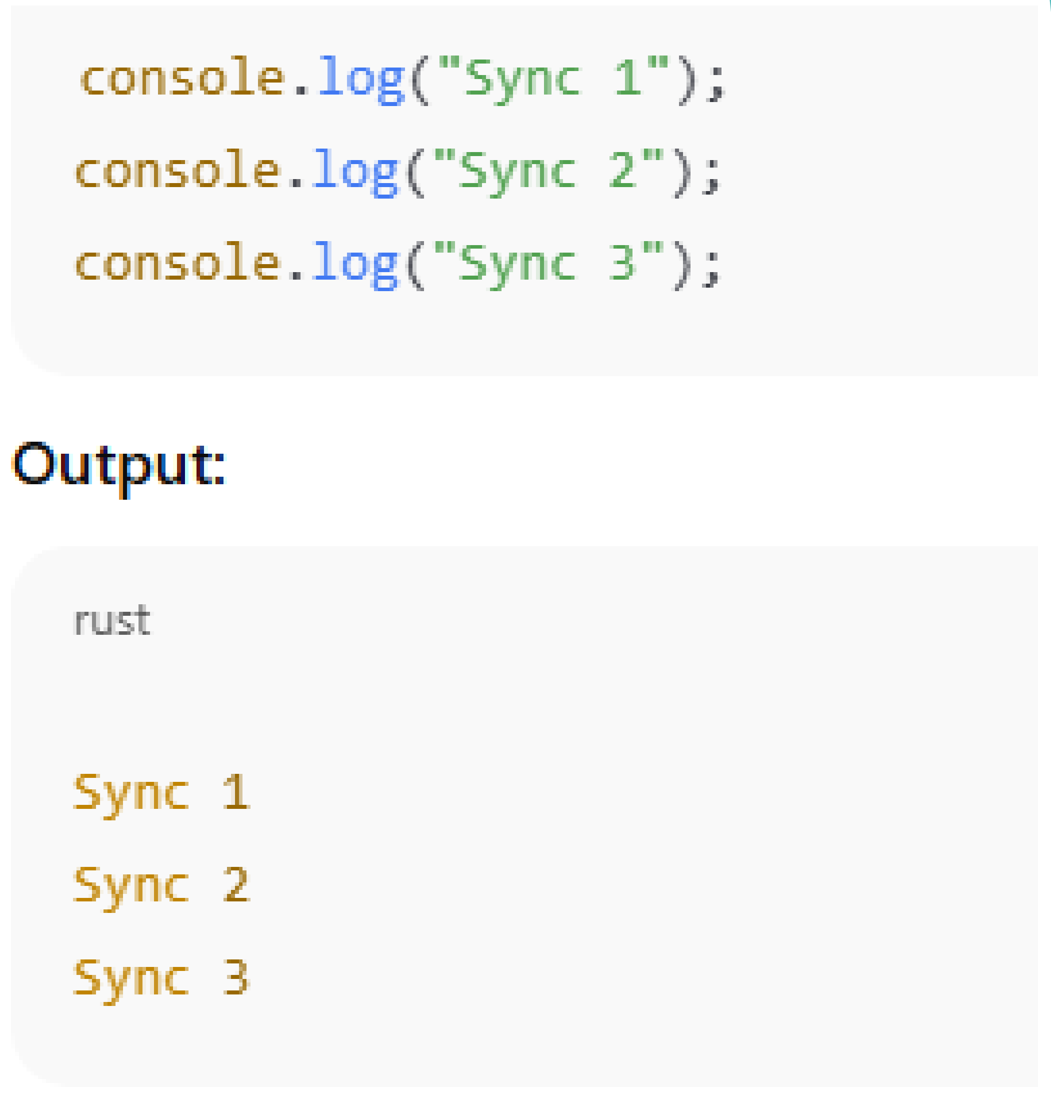
vbnet

```
Step 1
Step 2
Step 3
```

Async vs Sync Functions

Synchronous Functions (Sync):

- Executed line by line
- Each function waits for the previous to finish



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

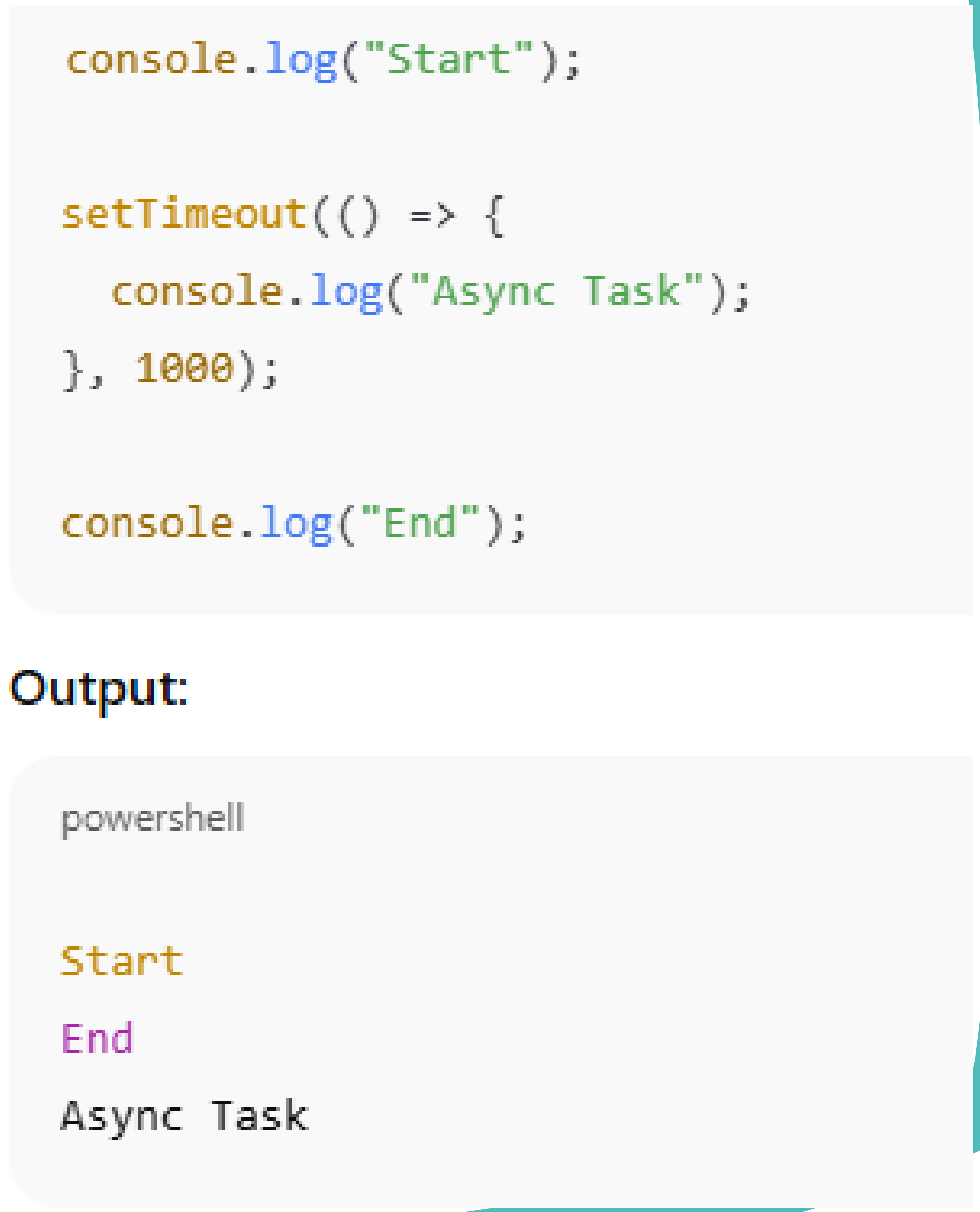
Output:

```rust
Sync 1
Sync 2
Sync 3
```

Async vs Sync Functions

Asynchronous Functions (Async):

- Can execute later, without blocking the rest of the code
- Examples: setTimeout, fetch, Promises, async/await



```javascript
console.log("Start");

setTimeout(() => {
  console.log("Async Task");
}, 1000);

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

Output:

```powershell
Start
End
Async Task
```

✅ Summary Table

Concept	Sync/Async	Example
Call Stack	Sync	Function calls
Event Queue	Async	setTimeout, fetch, events
Events	Async	click, input, load
Callback	Async	Functions passed to other functions
Callback Hell	Async	Nested setTimeouts or callbacks
Async vs Sync	Both	setTimeout vs console.log

Thank You