Software Engineering

Sync vs. Async Programming

JavaScript

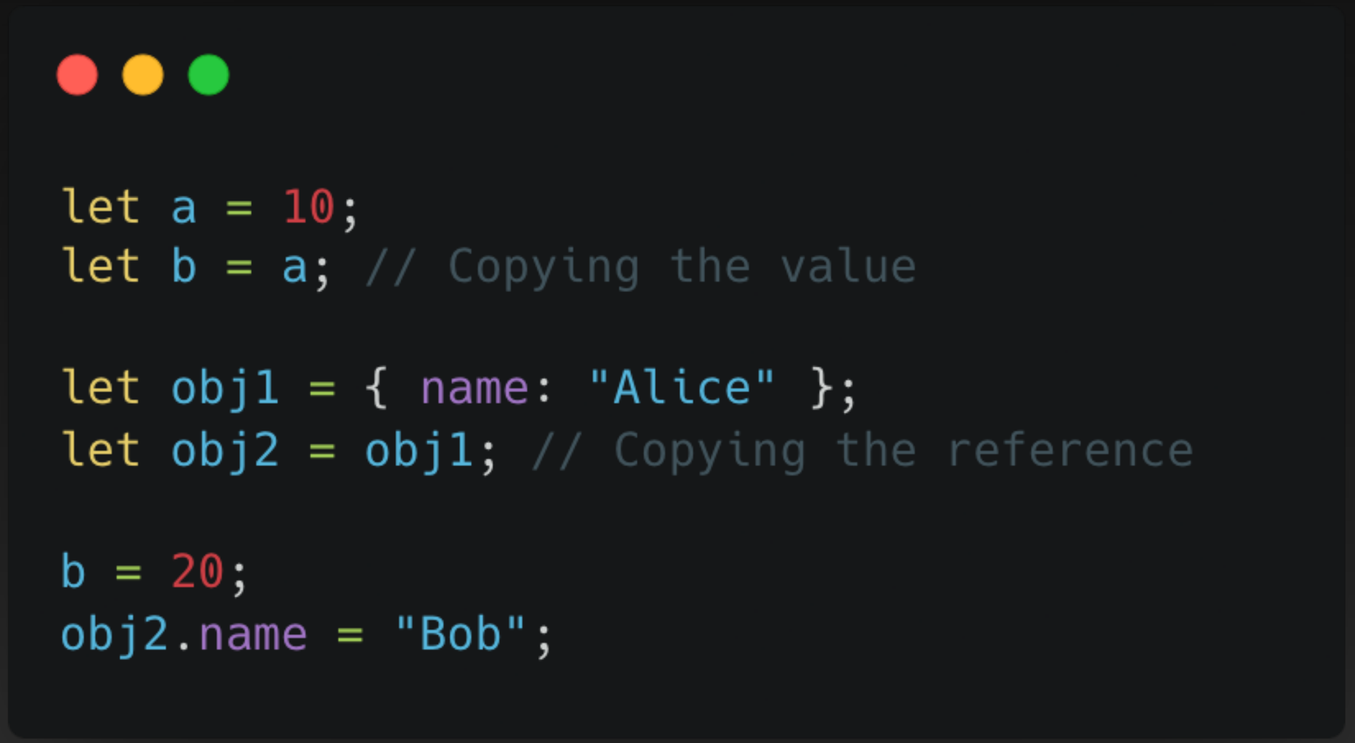
By: Yousef Gilany

Agenda Items

- Introduction to JavaScript Execution Model
- Introduction to Asynchronous Programming
- Asynchronous Patterns in JavaScript
  - Callback
  - Promises
  - Async/Await

Do Objects Differ from Variables in Memory?

Do Objects Differ from Variables in Memory?



```
let a = 10;
let b = a; // Copying the value

let obj1 = { name: "Alice" };
let obj2 = obj1; // Copying the reference

b = 20;
obj2.name = "Bob";
```

Do Objects Differ from Variables in Memory?

Expect the output

```
let a = 10;
let b = a; // Copying the value

let obj1 = { name: "So3aad" };
let obj2 = obj1; // Copying the reference

b = 20;
obj2.name = "Hussein";
```

```
console.log(a);

console.log(b);

console.log(obj1.name);

console.log(obj2.name);
```

Do Objects Differ from Variables in Memory?

Expect the output

```
let a = 10;
let b = a; // Copying the value

let obj1 = { name: "So3aad" };
let obj2 = obj1; // Copying the reference

b = 20;
obj2.name = "Hussein";
```

```
console.log(a);
// 10 (unchanged, because it's a copy)
console.log(b);
// 20 (new value assigned)
console.log(obj1.name);
// "Hussein" (modified through obj2)
console.log(obj2.name);
// "Hussein" (same reference as obj1)
```

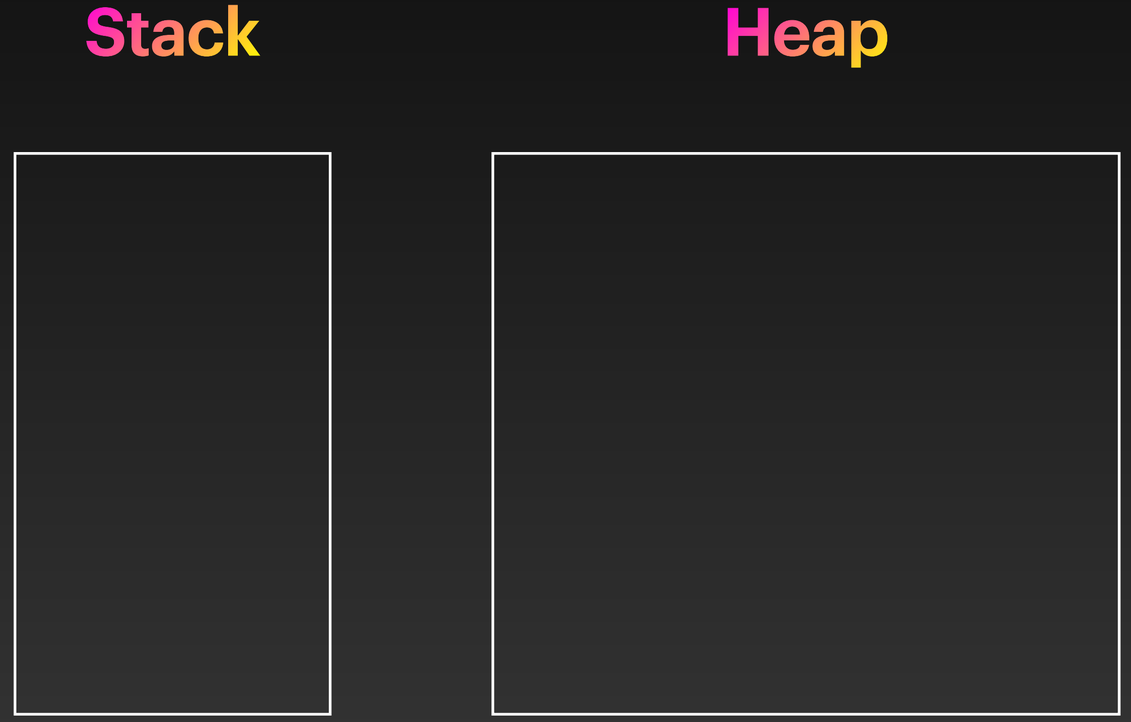
Visual explanation

Visual explanation

```
let a = 10;
let b = a;

let obj1 = { name: "So3aad" };
let obj2 = obj1;

b = 20;
obj2.name = "Hussein";
```

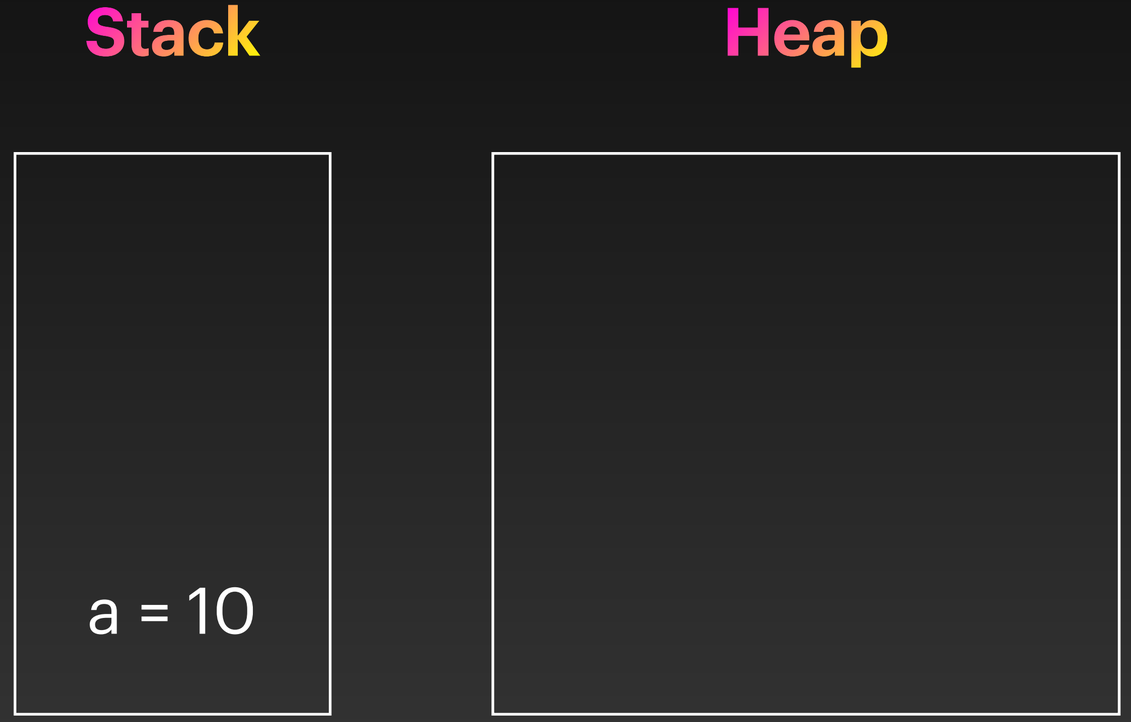


Visual explanation

```
let a = 10;
let b = a;

let obj1 = { name: "So3aad" };
let obj2 = obj1;

b = 20;
obj2.name = "Hussein";
```

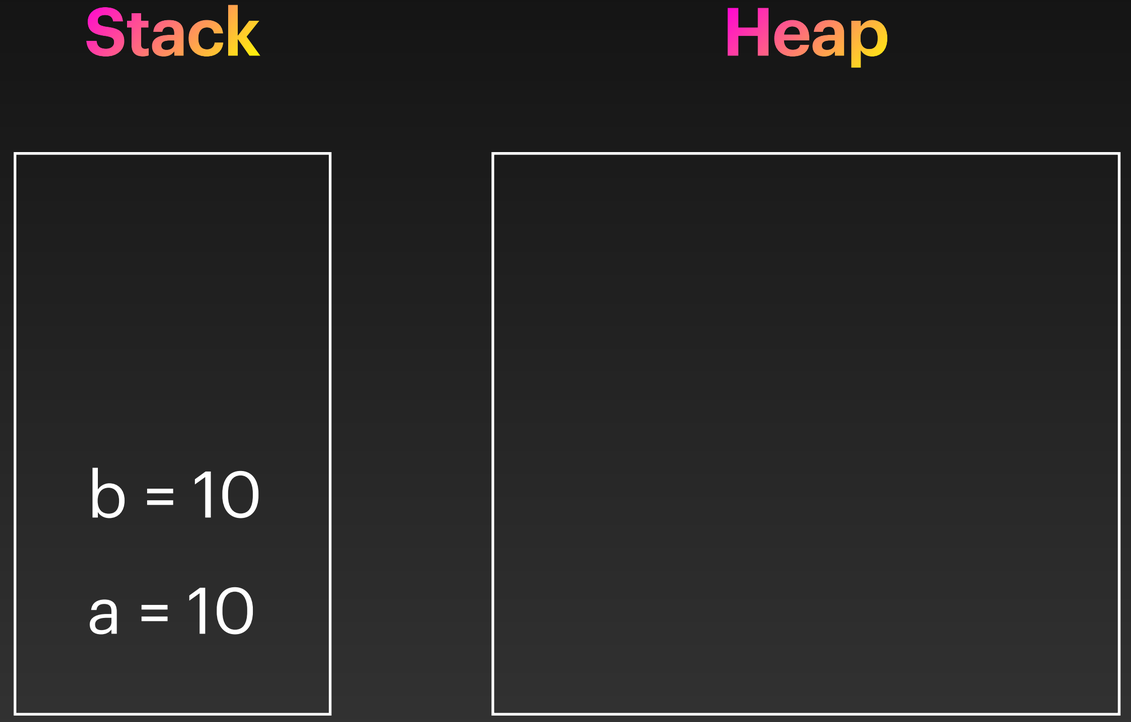


Visual explanation

```
let a = 10;
let b = a;

let obj1 = { name: "So3aad" };
let obj2 = obj1;

b = 20;
obj2.name = "Hussein";
```

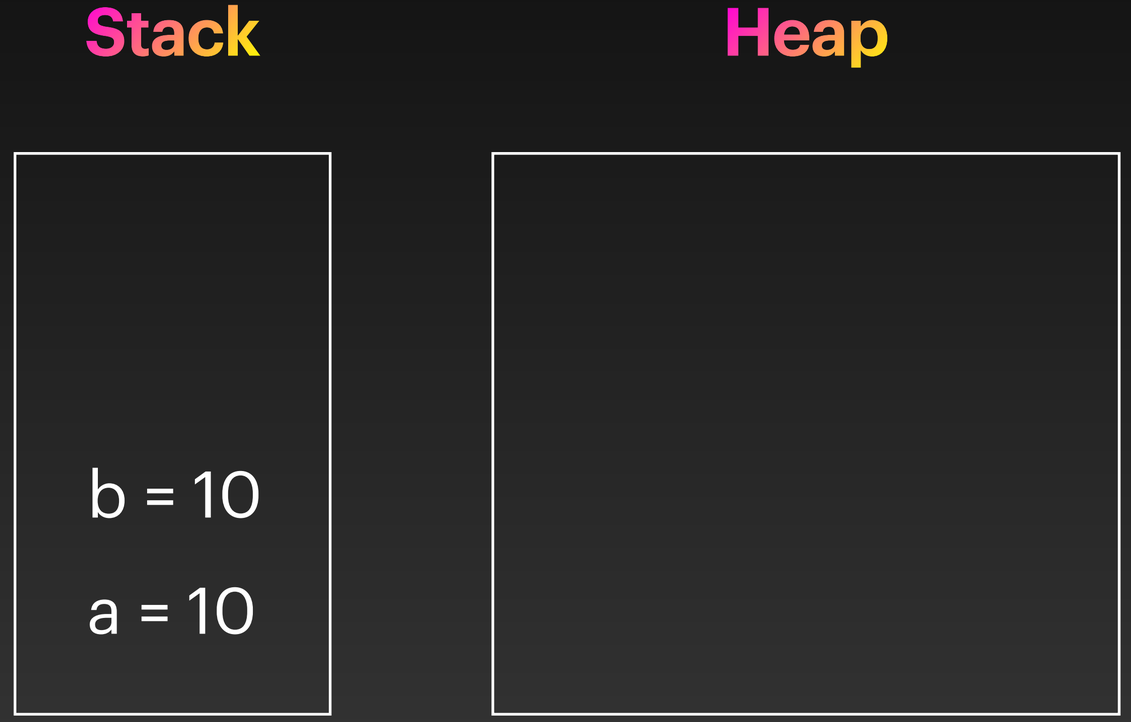


Visual explanation

```
let a = 10;
let b = a;

let obj1 = { name: "So3aad" };
let obj2 = obj1;

b = 20;
obj2.name = "Hussein";
```



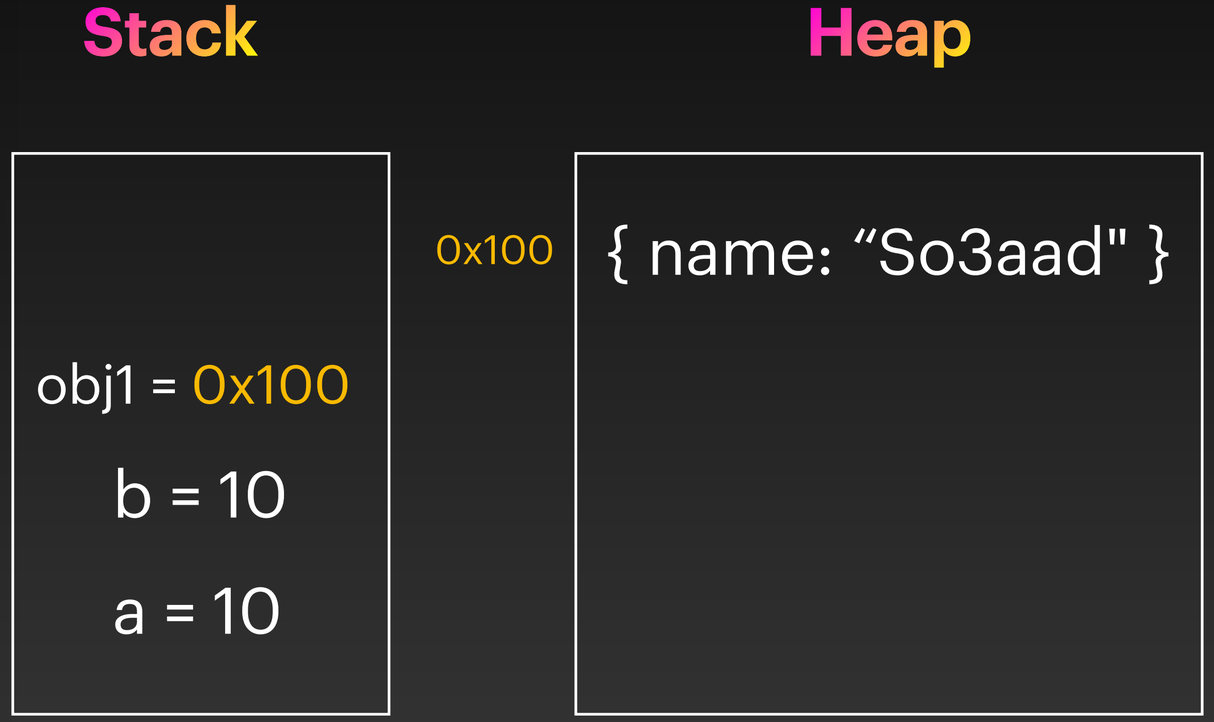
✅ Primitives are stored in the stack

Visual explanation

```
let a = 10;
let b = a;

let obj1 = { name: "So3aad" };
let obj2 = obj1;

b = 20;
obj2.name = "Hussein";
```

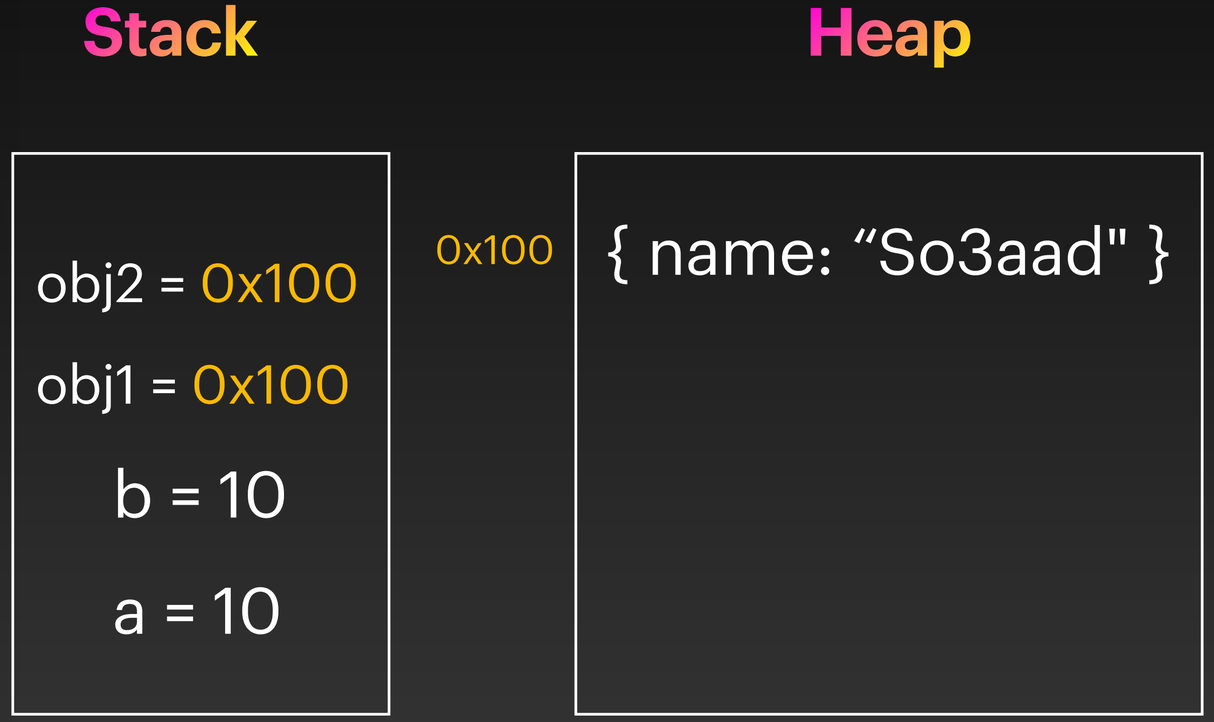


Visual explanation

```
let a = 10;
let b = a;

let obj1 = { name: "So3aad" };
let obj2 = obj1;

b = 20;
obj2.name = "Hussein";
```



Feature	Primitive (Stack)	Object (Heap)
Storage	Directly in the stack	Stored in heap, referenced from stack
Copy Behavior	Copies value	Copies reference (not value)
Mutability	Immutable (new copy created when changed)	Mutable (changes reflect across all references)

Introduction to JavaScript Execution Model

Example 1

Callback Queue in JavaScript

Introduction to JavaScript Execution Model

Callback Queue in JavaScript

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

setTimeout(() => {
    console.log("Callback in queue");
}, 0);

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

Introduction to JavaScript Execution Model

Callback Queue in JavaScript

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

setTimeout(() => {
    console.log("Callback in queue");
}, 0);

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

1. "Start" is printed.
2. setTimeout is called with 0ms delay, but it does not execute immediately. Instead, the callback is placed in the callback queue.
3. "End" is printed.
4. The event loop checks the call stack. Once it is empty, it moves the callback from the queue to the stack.
5. "Callback in queue" is printed.

Example 2

Callback Queue in JavaScript

Introduction to JavaScript Execution Model

Callback Queue in JavaScript

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

setTimeout(() => {
    console.log("Timeout 1");
}, 1000);

setTimeout(() => {
    console.log("Timeout 2");
}, 500);

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

Introduction to JavaScript Execution Model

Callback Queue in JavaScript

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

setTimeout(() => {
    console.log("Timeout 1");
}, 1000);

setTimeout(() => {
    console.log("Timeout 2");
}, 500);

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

1. "Start" is printed.
2. setTimeout(..., 1000) places Timeout 1 in the callback queue after 1000ms.
3. setTimeout(..., 500) places Timeout 2 in the callback queue after 500ms.
4. "End" is printed.
5. After 500ms, Timeout 2 moves to the callback queue and executes when the call stack is empty.
6. After 1000ms, Timeout 1 moves to the callback queue and executes.

Callback Queue (Macrotask Queue)

vs.

Job Queue (Microtask Queue)

JavaScript has **two** types of queues for handling asynchronous tasks

JavaScript has **two** types of queues for handling asynchronous tasks

Feature	Callback Queue (Macrotask Queue)	Job Queue (Microtask Queue)
Examples	setTimeout(), setInterval(), setImmediate()	Promise.then(), catch(), finally()
Execution Order	Executes **after** microtasks	Executes **before** macrotasks
Priority	Lower	Higher
Processed	After the job queue is empty	After every synchronous task and before the next macrotask
Execution Timing	Runs after microtasks are finished	Runs immediately after the call stack is empty

Example 1

Job Queue in JavaScript

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

setTimeout(() => {
    console.log("Macrotask: setTimeout");
}, 0);

Promise.resolve().then(() => {
    console.log("Microtask: Promise");
});

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

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

setTimeout(() => {
    console.log("Macrotask: setTimeout");
}, 0);

Promise.resolve().then(() => {
    console.log("Microtask: Promise");
});

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

1. "Start" is printed.
2. setTimeout(..., 0) schedules its callback in the callback queue (macrotask queue).
3. Promise.resolve().then(...) schedules its callback in the job queue (microtask queue).
4. "End" is printed.
5. The microtask queue runs first → "Microtask: Promise" is printed.
6. The callback queue runs next → "Macrotask: setTimeout" is printed.

Example 2

Job Queue in JavaScript

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

setTimeout(() => {
    console.log("Macrotask: setTimeout");

    Promise.resolve().then(() => {
        console.log("Microtask inside
Macrotask");
    });

}, 0);

Promise.resolve().then(() => {
    console.log("Microtask: Promise 1");
});

Promise.resolve().then(() => {
    console.log("Microtask: Promise 2");
});

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

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

setTimeout(() => {
    console.log("Macrotask: setTimeout");

    Promise.resolve().then(() => {
        console.log("Microtask inside
Macrotask");
    });

}, 0);

Promise.resolve().then(() => {
    console.log("Microtask: Promise 1");
});

Promise.resolve().then(() => {
    console.log("Microtask: Promise 2");
});

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

1. "Start" is printed.
2. setTimeout callback goes into the callback queue.
3. Promise.then callbacks go into the job queue.
4. "End" is printed.
5. The **microtask queue** runs first:
   - "Microtask: Promise 1" is printed.
   - "Microtask: Promise 2" is printed.
6. The **callback queue** runs next:
   - "Macrotask: setTimeout" is printed.
   - A new microtask is added (Promise.then inside setTimeout).
   - The microtask "Microtask inside Macrotask" runs immediately before another macrotask.

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

setTimeout(() => {
    console.log("Macrotask: setTimeout");

    Promise.resolve().then(() => {
        console.log("Microtask inside
Macrotask");
    });

}, 0);

Promise.resolve().then(() => {
    console.log("Microtask: Promise 1");
});

Promise.resolve().then(() => {
    console.log("Microtask: Promise 2");
});

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

1. "Start" is printed.
2. setTimeout callback goes into the callback queue.
3. Promise.then callbacks go into the job queue.
4. "End" is printed.
5. The **microtask queue** runs first:
   - "Microtask: Promise 1" is printed.
   - "Microtask: Promise 2" is printed.
6. The **callback queue** runs next:
   - "Macrotask: setTimeout" is printed.
   - A new microtask is added (Promise.then inside setTimeout).
   - The microtask "Microtask inside Macrotask" runs immediately before another macrotask.

✅ Even inside a macrotask, microtasks execute before the next macrotask!

Introduction to Asynchronous Programming

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

let fetchedData = null;

async function fetchData() {
    let response = await
fetch("https://jsonplaceholder.typicode.com/todos/1");
    fetchedData = await response.json();
}

fetchData();

console.log("Data received:", fetchedData);

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

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

let fetchedData = null;

async function fetchData() {
    let response = await
fetch("https://jsonplaceholder.typicode.com/todos/1");
    fetchedData = await response.json();
}

fetchData();

console.log("Data received:", fetchedData);

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

Start fetching data...

Data received: null ❌

End of script.

Let’s try to solve the issue

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

let isDone = false;
let fetchedData = null;

async function fetchData() {
    let response = await
fetch("https://jsonplaceholder.typicode.com/todos/1");
    fetchedData = await response.json();
    isDone = true;
}

fetchData();

while (!isDone) {}

console.log("Data received:", fetchedData);
console.log("End of script.");
```

Start fetching data...

Data received: [OBJECT] ✅

End of script.

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

let isDone = false;
let fetchedData = null;

async function fetchData() {
    let response = await
fetch("https://jsonplaceholder.typicode.com/todos/1");
    fetchedData = await response.json();
    isDone = true;
}

fetchData();

while (!isDone) {}

console.log("Data received:", fetchedData);
console.log("End of script.");
```

Start fetching data...

Data received: [OBJECT] ✅

End of script.

It works, but it's a very bad practice.

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

let isDone = false;
let fetchedData = null;

async function fetchData() {
    let response = await
fetch("https://jsonplaceholder.typicode.com/todos/1");
    fetchedData = await response.json();
    isDone = true;
}

fetchData();

while (!isDone) {}

console.log("Data received:", fetchedData);
console.log("End of script.");
```

Start fetching data...

Data received: [OBJECT] ✅

End of script.

It works, but it's a very bad practice.

- The while (!isDone) {} loop freezes the event loop.
- The browser stops processing other tasks until isDone is true.
- This makes the page unresponsive.

So, what do we do?

So, what do we do?
We use Promises

Promise Maker

```
function maker(){
  return new Promise();
}
```

Promise Receiver

```
let promise = maker();
```

Promise Maker

```
function maker(){
  return new Promise();
}
```

Promise Maker

```
function maker(){
  return new Promise(function(resolve,reject) {
    setTimeout(function(){
    resolve("data")
    })
  });
}
```

Promise Maker

Promise Receiver

```
function maker(){
  return new Promise(function(resolve,reject) {
    setTimeout(function(){
    resolve("data")
    })
  });
}
```

```
let promise = maker();
```

Promise Maker

Promise Receiver

```
function maker(){
  return new Promise(function(resolve,reject) {
    setTimeout(function(){
    resolve("data")
    })
  });
}
```

```
const promise = maker();
promise.then(
function(data) {
  console.log("first param")
},
function(data) {
  console.log("second param")
})
```

But this is ugly code, right?

Promise Maker

Promise Receiver

```
function maker(){
  return new Promise(function(resolve,reject) {
    setTimeout(function(){
    resolve("data")
    })
  });
}
```

```
const promise = maker();
promise.then(
function(data) {
  console.log("first param")
},
function(data) {
  console.log("second param")
})
```

Promise Maker

Promise Receiver

```
function maker(){
  return new Promise(function(resolve,reject) {
    setTimeout(function(){
      if (true){
        resolve("data")
      } else {
        reject("error")
      }
    }, 2000)
  });
}
```

```
function onSuccess(data) {
  console.log("Got data")
}

function onFail(error) {
  console.log("Error happened")
}

maker().then(onSuccess,onFail)
```

What did we get from all of this?

What did we get from all of this?

1️⃣ Avoids Callback Hell

2️⃣ Better Error Handling (.catch())

3️⃣ Chaining Multiple Async Operations

4️⃣ Parallel Execution with Promise.all()

1️⃣ Avoids Callback Hell

```
function getUser(userId, callback) {
    setTimeout(() => {
        console.log("Fetched user");
        callback({ id: userId, name: "John Doe" });
    }, 1000);
}

function getPosts(userId, callback) {
    setTimeout(() => {
        console.log("Fetched posts");
        callback([{ id: 101, title: "Post 1" },
                  { id: 102, title: "Post 2" }]);
    }, 1000);
}

function getComments(postId, callback) {
    setTimeout(() => {
        console.log(`Fetched comments for post ${postId}`);
        callback([{ id: 201, text: "Nice post!" },
                  { id: 202, text: "Great read!" }]);
    }, 1000);
}
```

```
getUser(1, (user) => {
    console.log("User:", user);

    getPosts(user.id, (posts) => {
        console.log("Posts:", posts);

        getComments(posts[0].id, (comments) => {
            console.log("Comments:", comments);

            getComments(posts[1].id, (moreComments) => {
                console.log("Comments:", moreComments);

                console.log("Done!");
            });
        });
    });
});
```

1️⃣ Avoids Callback Hell

```
function getUser(userId, callback) {
    setTimeout(() => {
        console.log("Fetched user");
        callback({ id: userId, name: "John Doe" });
    }, 1000);
}

function getPosts(userId, callback) {
    setTimeout(() => {
        console.log("Fetched posts");
        callback([{ id: 101, title: "Post 1" },
                  { id: 102, title: "Post 2" }]);
    }, 1000);
}

function getComments(postId, callback) {
    setTimeout(() => {
        console.log(`Fetched comments for post ${postId}`);
        callback([{ id: 201, text: "Nice post!" },
                  { id: 202, text: "Great read!" }]);
    }, 1000);
}
```

Ugly code

1️⃣ Avoids Callback Hell

```
function getUser(userId, callback) {
    setTimeout(() => {
        console.log("Fetched user");
        callback({ id: userId, name: "John Doe" });
    }, 1000);
}

function getPosts(userId, callback) {
    setTimeout(() => {
        console.log("Fetched posts");
        callback([{ id: 101, title: "Post 1" },
                  { id: 102, title: "Post 2" }]);
    }, 1000);
}

function getComments(postId, callback) {
    setTimeout(() => {
        console.log(`Fetched comments for post ${postId}`);
        callback([{ id: 201, text: "Nice post!" },
                  { id: 202, text: "Great read!" }]);
    }, 1000);
}
```

```
getUser(1)
    .then(user => getPosts(user.id))
    .then(posts => getComments(posts[0].id))
    .then(comments => console.log("Done!"));
```

What did we get from all of this?

1️⃣ Avoids Callback Hell

2️⃣ Better Error Handling (.catch())

3️⃣ Chaining Multiple Async Operations

4️⃣ Parallel Execution with Promise.all()

2️⃣ Better Error Handling (.catch())

```
getUser(1)
    .then(user => getPosts(user.id))
    .then(posts => getComments(posts[0].id))
    .then(comments => console.log("Done!"));
```

2️⃣ Better Error Handling (.catch())

```
getUser(1)
    .then(user => getPosts(user.id))
    .then(posts => getComments(posts[0].id))
    .then(comments => console.log("Done!"));
```

```
getUser(1)
    .then(user => getPosts(user.id))
    .then(posts => getComments(posts[0].id))
    .then(comments => console.log("Done!"))
    .catch(error => console.error("Error:", error));
```

What did we get from all of this?

1️⃣ Avoids Callback Hell

2️⃣ Better Error Handling (.catch())

3️⃣ Chaining Multiple Async Operations

4️⃣ Parallel Execution with Promise.all()

3️⃣ Chaining Multiple Async Operations

```
getUser(1)
    .then(user => getPosts(user.id))
    .then(posts => getComments(posts[0].id))
    .then(comments => console.log("Done!"))
    .catch(error => console.error("Error:", error));
```

4️⃣ Parallel Execution with Promise.all()

```
getUser(1)
    .then(user => getPosts(user.id))
    .then(posts => getComments(posts[0].id))
    .then(comments => console.log("Done!"))
    .catch(error => console.error("Error:", error));
```

4️⃣ Parallel Execution with Promise.all()

```
getUser(1)
    .then(user => getPosts(user.id))
    .then(posts => getComments(posts[0].id))
    .then(comments => console.log("Done!"))
    .catch(error => console.error("Error:", error));
```

```
getUser(1)
    .then(user => getPosts(user.id))
    .then(posts => Promise.all([
    getComments(posts[0].id),
    getComments(posts[1].id)]))
    .catch(error => {});
```

async/await

async/await

```
async function fetchData() {
    try {
        let user = await getUser(1);
        let posts = await getPosts(user.id);
        let comments = await getComments(posts[0].id);
        console.log("Done!");
    } catch (error) {
        console.error("Error:", error);
    }
}

fetchData();
```

Both Promises and async/await offer excellent performance. However, in some cases, Promises can be slightly cleaner, depending on the complexity.

Thank you