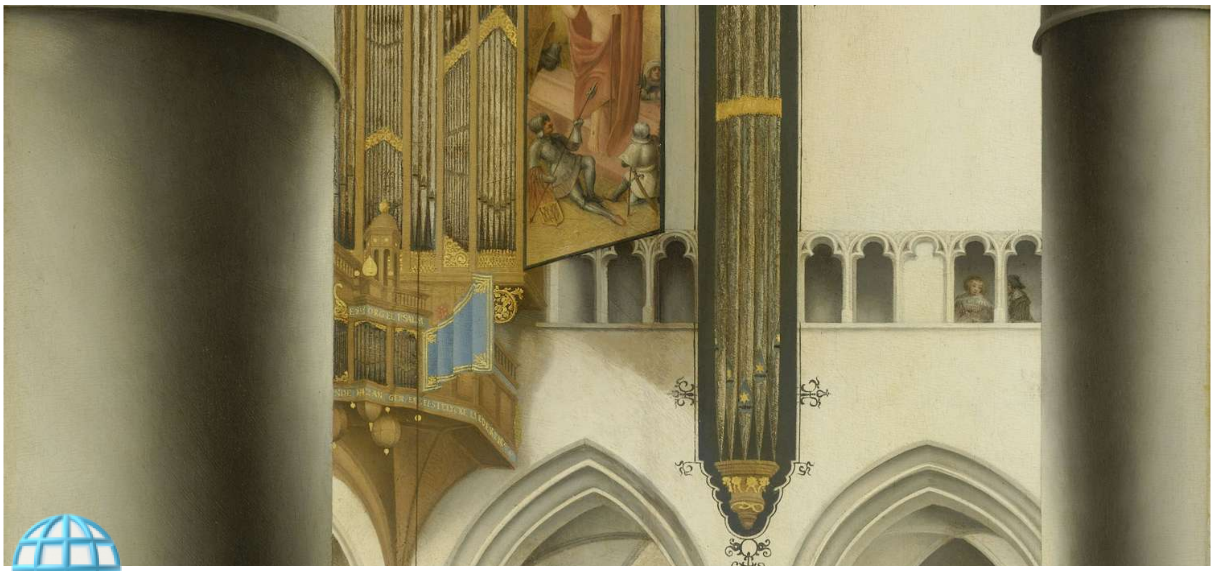


JavaScript - Synchronous vs Asynchronous

Baraneetharan Ramasamy

Task, Process, Thread, Synchronous, Asynchronous, Concurrent, and Parallel

1. Task

- A task refers to a unit of work or an operation that needs to be performed.
- It can represent any computational activity, such as reading a file, performing calculations, or making a network request.

2. Process

- A process is an instance of a running program.
- It has its own memory space, resources, and execution context.
- Processes can run independently and communicate with each other through inter-process communication (IPC).

3. Thread

- A thread is the smallest unit of execution within a process.
- Multiple threads can exist within a single process, sharing the same memory space.
- Threads allow concurrent execution of tasks within the same process.

4. Synchronous (Sync)

- In synchronous programming, tasks are executed sequentially, one after the other.
- When you call a synchronous function, your program waits for it to complete before moving on to the next task.
- Imagine standing in a queue—you wait for your turn before proceeding.

5. Asynchronous (Async)

- Asynchronous programming allows tasks to run concurrently without blocking the main execution thread.
- When you call an asynchronous function, it doesn’t wait for the result immediately.
- Instead, it continues executing other tasks while waiting for the asynchronous operation to complete.
- Think of it like ordering food online—you can continue browsing while waiting for the delivery.

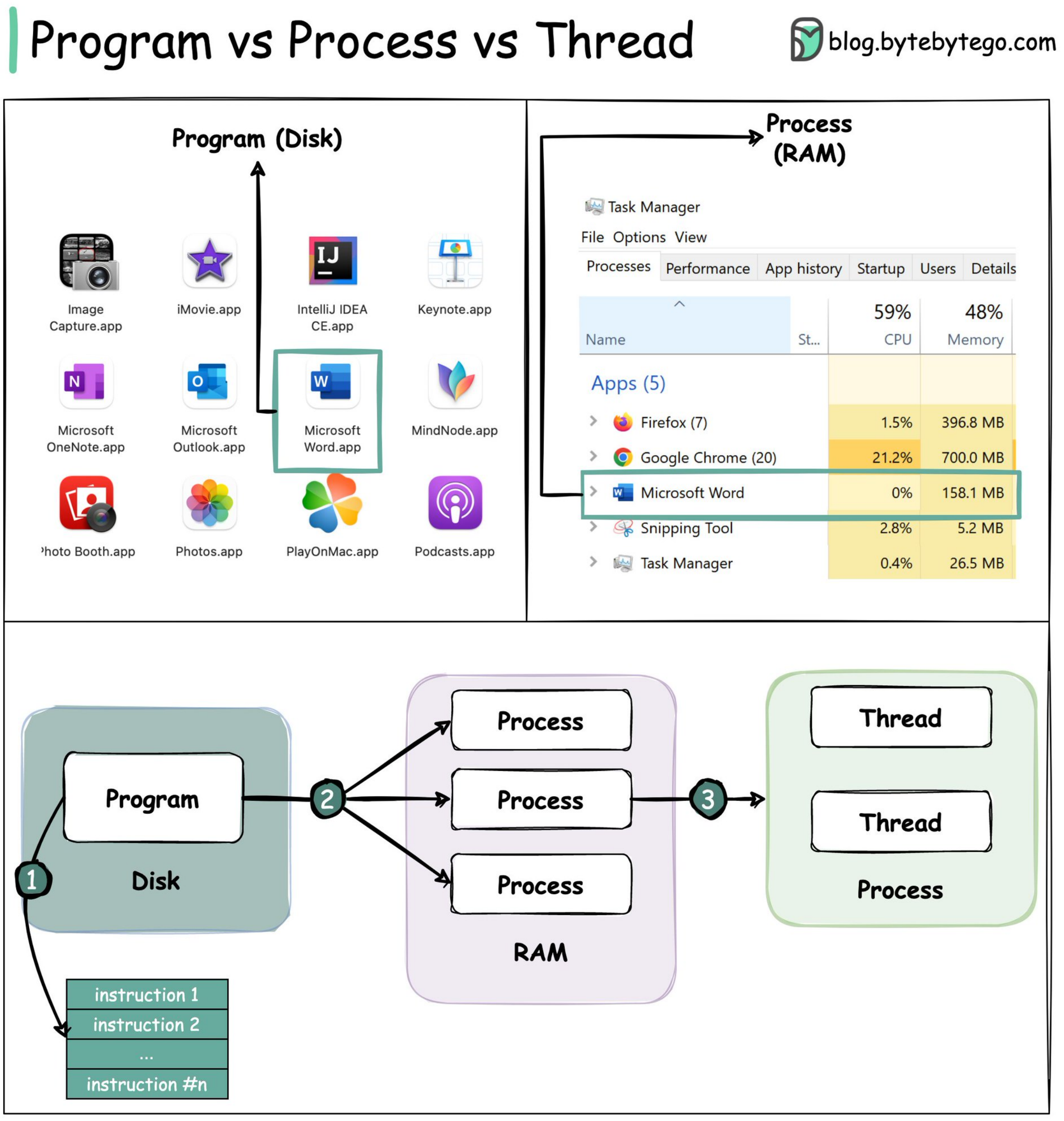
6. Concurrent

- Concurrency involves managing multiple tasks that can potentially overlap in execution time.
- These tasks may not necessarily run simultaneously but can share the same execution context (e.g., a single CPU core).
- Imagine juggling multiple balls—you handle them one by one, but they appear to be happening concurrently.

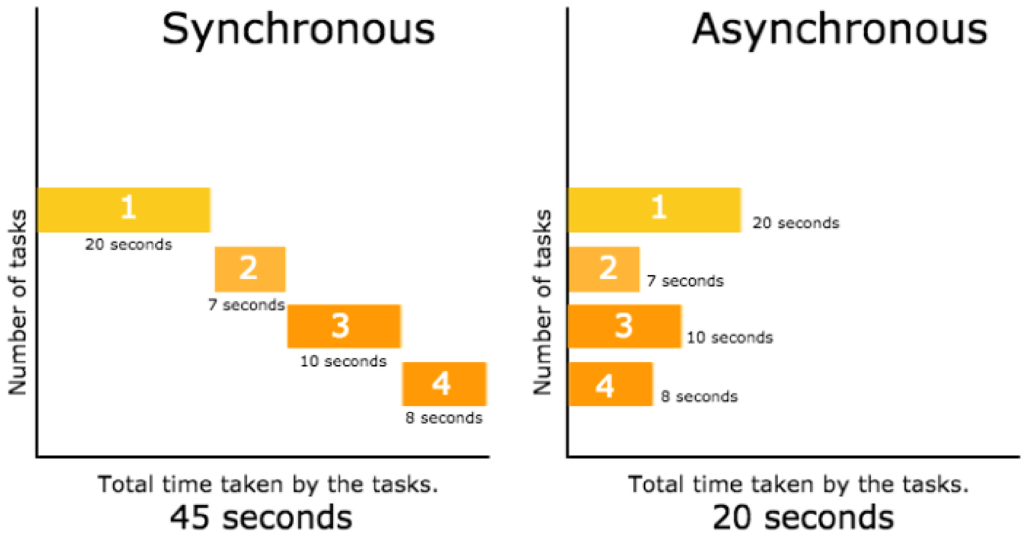
7. Parallel

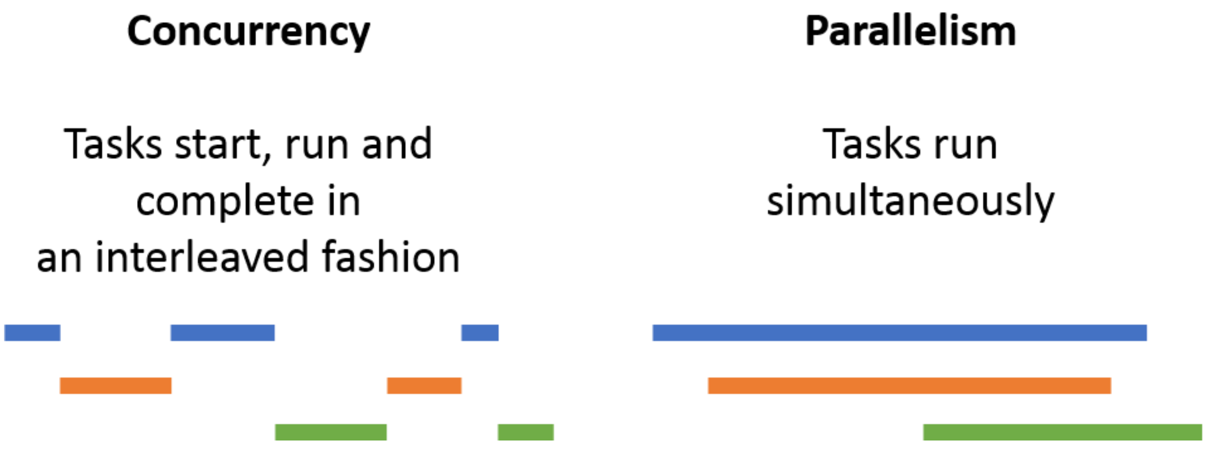
- Parallelism involves executing tasks simultaneously using multiple cores or processors.

- It's like having multiple jugglers, each handling a different set of balls at the same time.



Name	St...	59% CPU	48% Memory
Apps (5)			
Firefox (7)		1.5%	396.8 MB
Google Chrome (20)		21.2%	700.0 MB
Microsoft Word		0%	158.1 MB
Snipping Tool		2.8%	5.2 MB
Task Manager		0.4%	26.5 MB





Tasks start, run and complete in an interleaved fashion

Tasks run simultaneously

Stack and Heap memory in Java

What is stored where?

Stack :

– local variables

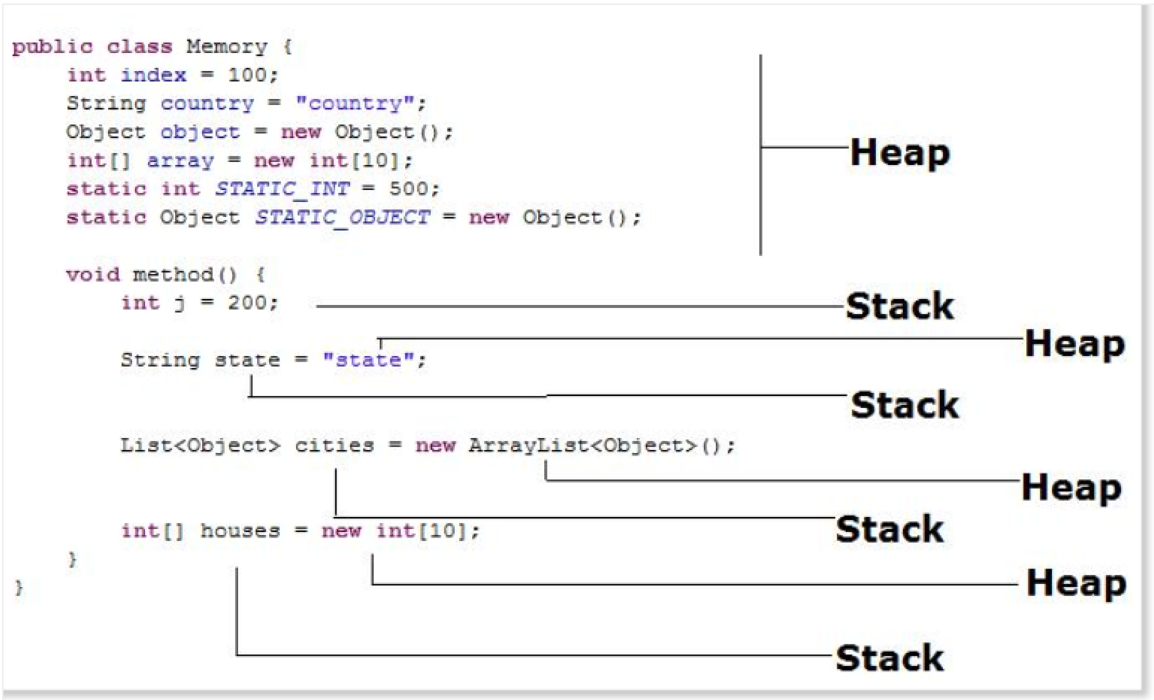
Heap :

– instance variables

– static variables

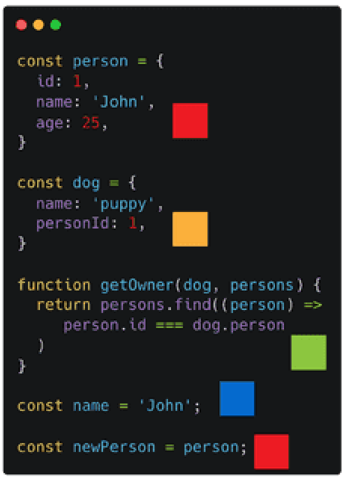
– objects

Following snippet will help to understand better.

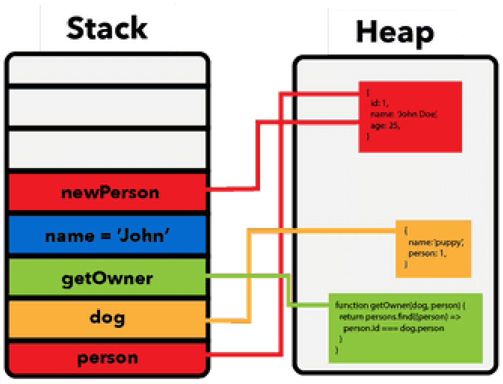


```
public class Memory {
    int index = 100;
    String country = "country";
    Object object = new Object();
    int[] array = new int[10];
    static int STATIC_INT = 500;
    static Object STATIC_OBJECT = new Object();

    void method() {
        int j = 200;

        String state = "state";

        List<Object> cities = new ArrayList<Object>();

        int[] houses = new int[10];
    }
}
```

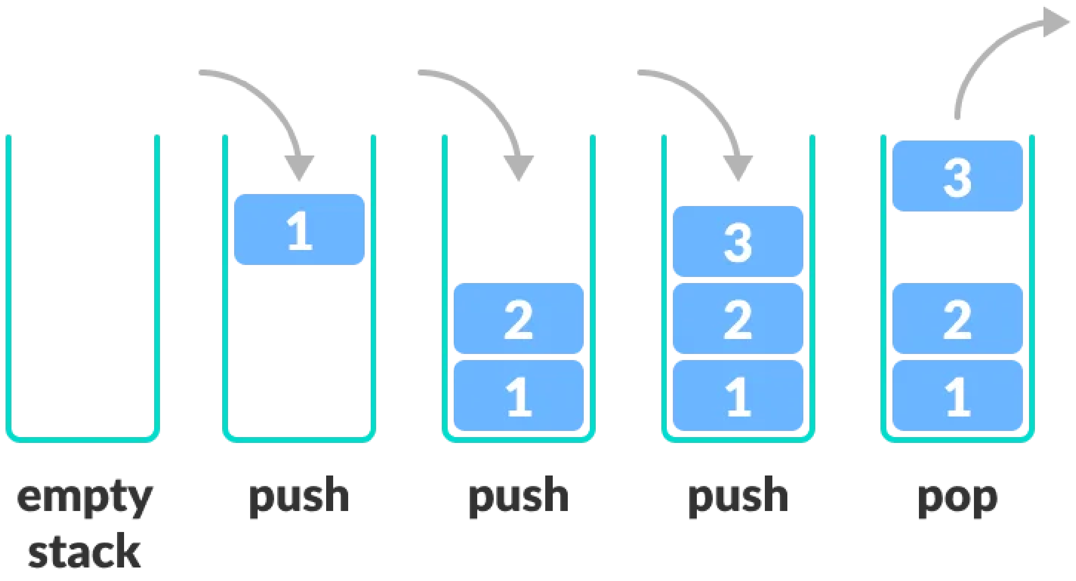
Stack and Heap memory in JavaScript



```
const person = {
  id: 1,
  name: 'John',
  age: 25,
}

const dog = {
  name: 'puppy',
  personId: 1,
}

function getOwner(dog, persons) {
  return persons.find((person) =>
    person.id === dog.person
  )
}

const name = 'John';

const newPerson = person;
```


Stack
Heap
newPerson
name = 'John'
getOwner
dog
person
{ id: 1, name: 'John Doe', age: 25, }
{ name:'puppy', person: 1, }
function getOwner(dog, person) { return persons.find((person) => person.id === dog.person) }

What is Stack?



What is JavaScript?

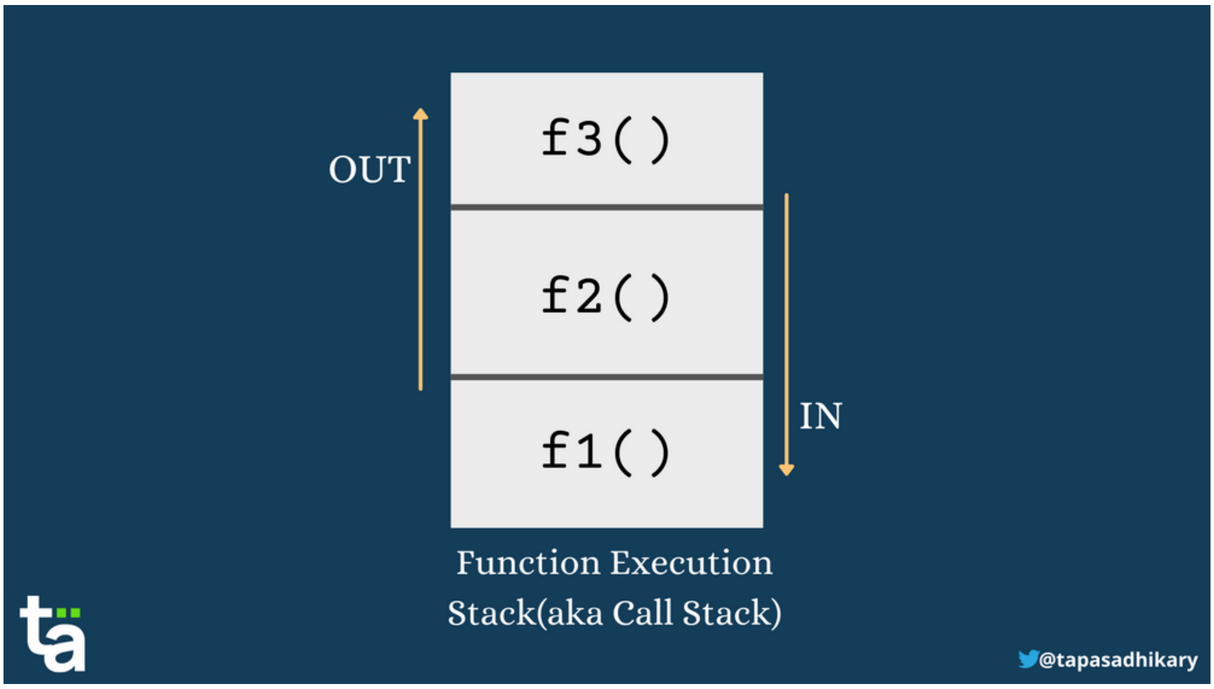
JavaScript is a single-threaded, non-blocking, asynchronous, concurrent programming language with lots of flexibility.

1. **Single-Threaded:**

- JavaScript runs on a single thread, meaning it processes one task at a time.
- This simplicity makes it easier to implement and avoids complex issues like deadlocks that can occur in multi-threaded environments.
- However, being single-threaded doesn’t mean JavaScript can’t handle multiple tasks; it just does so sequentially.

2. **Non-Blocking:**
   - Although JavaScript is single-threaded, it can still handle asynchronous tasks efficiently.
   - Asynchronous operations (like fetching data from a server or reading files) don’t block the main thread.
   - Instead, JavaScript schedules these tasks and continues executing other code while waiting for the asynchronous operation to complete.
   - When the result is ready, it triggers a callback or resolves a promise.
   - This non-blocking behavior allows web applications to remain responsive even during time-consuming tasks.
3. **Concurrent:**
   - JavaScript achieves concurrency through its event loop.
   - While it processes tasks sequentially, it can switch between different tasks efficiently.
   - For example, when waiting for data from an API, JavaScript doesn’t halt everything else; it continues executing other code.
   - When the API response arrives, it handles it without blocking the main thread.
   - This concurrency model allows JavaScript to juggle multiple tasks without creating additional threads.

Call Stack in JavaScript



By default, every line in a function executes sequentially, one line at a time. The same is applicable even when you invoke multiple functions in your code. Again, line by line.

Example 1

```
function function1() {
    console.log('function1 called')
}
function function2() {
    console.log('function2 called')
}
function function3() {
    console.log('function3 called')
}

function1()
function2()
function3();
```

Output

```
function1 called
function2 called
function3 called
```

https://www.freecodecamp.org/news/content/images/2021/09/first-flow.gif

Example 2

```
function function1() {
    console.log('function1 called')
}
function function2() {
    function1()
    console.log('function2 called')
}
function function3() {
    function2()
    console.log('function3 called')
}

function3();
```

Output

```
function1 called
function2 called
function3 called
```

`function execution stack` is sequential. This is the `Synchronous` part of JavaScript.

How make JavaScript as asynchronous?

In JavaScript, a **callback** is a function that is passed as an argument to another function and is executed after the first function has completed its task. This allows for asynchronous execution, where you can perform operations that take time without blocking the main thread of execution.

callback function

```
function printMe() {
  console.log('print me');
}

setTimeout(printMe, 2000);
```

The `setTimeout` function executes a function after a certain amount of time has elapsed. In the code above, the text `print me` logs into the console after a delay of 2 seconds.

Built-in callback functions commonly used in the Browser API

- **setTimeout(callback, delay)**: Executes a callback after a specified delay.
- **setInterval(callback, interval)**: Repeatedly calls a callback at every given interval.
- **addEventListener(event, callback)**: Executes a callback when a specified event is triggered.
- **requestAnimationFrame(callback)**: Tells the browser to perform an animation and requests that the browser calls a specified callback to update an animation before the next repaint.
- **fetch(url).then(callback)**: The `fetch` API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses. It returns a Promise that resolves to the Response to that request, whether it is successful or not, and a callback can be added with `.then()`.
- **XMLHttpRequest.onreadystatechange**: An event handler that is called whenever the readyState attribute of the XMLHttpRequest changes; often used in AJAX requests.

Simple Callback Example

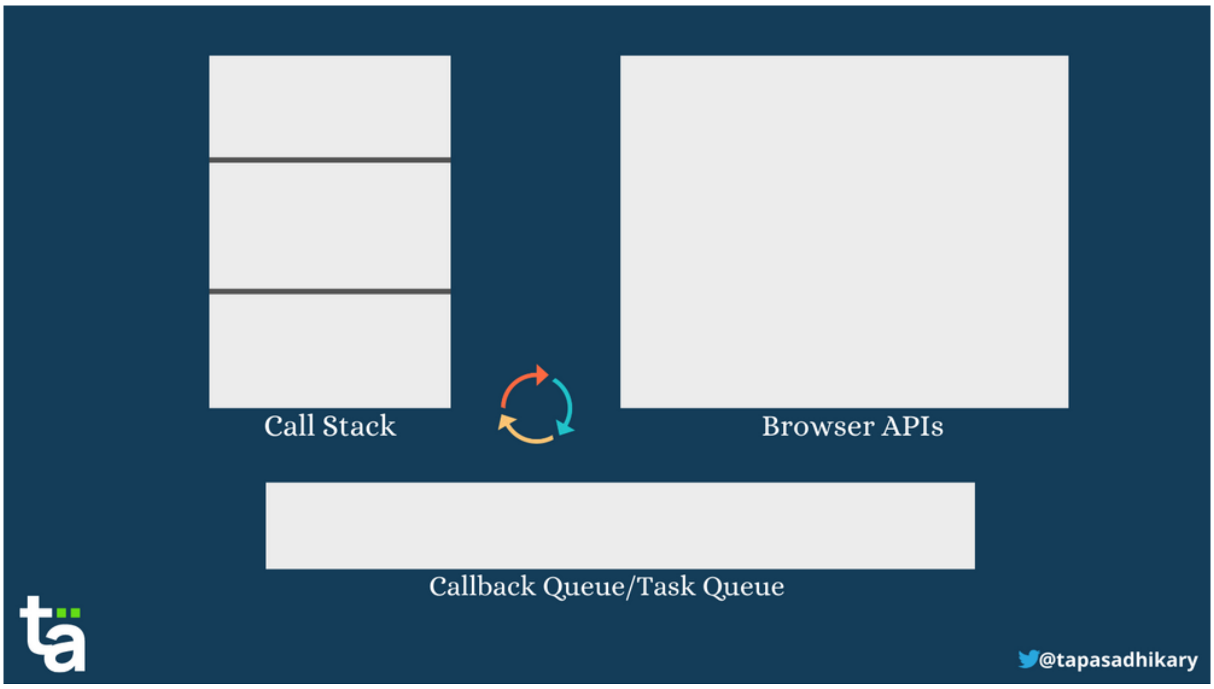
```
function printMe() {
  console.log('print me');
}

function test() {
  console.log('test');
}

setTimeout(printMe, 2000);
test();
```

Output

```
test
printMe
```

JavaScript Callback Queue Works (aka Task Queue)

JavaScript maintains a queue of callback functions. It is called a callback queue or task queue. A queue data structure is `First-In-First-Out(FIFO)`



Callback Example 2

```
function f1() {
    console.log('f1');
}

function f2() {
    console.log('f2');
}

function main() {
    console.log('main');

    setTimeout(f1, 0);

    f2();
}

main();
```

```
main
f2
f1
```

https://www.freecodecamp.org/news/content/images/2021/09/third-flow.gif

A callback function is like when you ask your mommy to make you a sandwich, and while she's making it, you go play with your toys. When the sandwich is ready, your mommy calls you back to the kitchen to eat it. The "calling back" part is like a callback function in programming - it's a way to tell the computer to do something (like make a sandwich) and then do something else (like play with toys) while waiting for the first thing to be done. When it's done, the computer "calls back" and lets you know that it's ready for the next step (like eating the sandwich).

Here is an example of a callback function in JavaScript that demonstrates the sandwich-making analogy:

```
function makeSandwich(ingredient1, ingredient2, callback) {
  console.log(`Making a sandwich with ${ingredient1} and
${ingredient2}...`);
  setTimeout(() => {
    const sandwich = `🥪(${ingredient1} & ${ingredient2})`;
    console.log(`Sandwich is ready: ${sandwich}`);
    callback(sandwich);
  }, 3000);
}
```

```
function eatSandwich(sandwich) {
  console.log(`Eating the sandwich: ${sandwich}`);
}

makeSandwich('🍅', '🧀', eatSandwich);
console.log('Playing with toys while waiting for the sandwi
ch...');
```

```
Making a sandwich with 🍅 and 🧀...
Playing with toys while waiting for the sandwich...
Sandwich is ready: 🥪(🍅 & 🧀)
Eating the sandwich: 🥪(🍅 & 🧀)
```

Callback hell

```
function makeSandwich(ingredient1, ingredient2, callback) {
    if (typeof ingredient1 !== 'string' || typeof ingredien
t2 !== 'string') {
      const error = new Error('Ingredients must be string
s');
      callback(null, error);
      return;
    }
    console.log(`Making a sandwich with ${ingredient1} and
${ingredient2}...`);
    setTimeout(() => {
      const sandwich = `🥪(${ingredient1} & ${ingredient2})
`;
      console.log(`Sandwich is ready: ${sandwich}`);
      callback(sandwich, null);
    }, 3000);
  }

  function cutVegetables(callback) {
    console.log("Cutting vegetables...");
    setTimeout(() => {
      console.log("Vegetables are ready.");
      callback();
    }, 2000);
  }

  function toastBread(callback) {
    console.log("Toasting bread...");
    setTimeout(() => {
      console.log("Bread is toasted.");
      callback();
    }, 1500);
  }

  function eatSandwich(sandwich) {
    console.log(`Eating the sandwich: ${sandwich}`);
  }

  try {
    toastBread(() => {
      cutVegetables(() => {
        makeSandwich('🍅', '🧀', (sandwich, error) => {
          if (error) {
            console.error(`An error occurred while making t
he sandwich: ${error}`);
          } else {
            eatSandwich(sandwich);
          }
        });
      });
    });
  } catch (error) {
    console.error(`An error occurred while making the sandw
ich: ${error}`);
  }
console.log('Playing with toys while waiting for the sandwi
ch...');
```

```
Toasting bread...
Playing with toys while waiting for the sandwich...
Bread is toasted.
Cutting vegetables...
Vegetables are ready.
Making a sandwich with 🍅 and 🧀...
Sandwich is ready: 🥪(🍅 & 🧀)
Eating the sandwich: 🥪(🍅 & 🧀)
```

In this updated example, we have added two additional asynchronous tasks: cutting vegetables and toasting bread. To ensure that these tasks are completed before making the sandwich, we have to nest the callbacks, resulting in a nested structure. This nesting can quickly become more complex as the number of asynchronous operations increases, leading to callback hell.

Callback hell makes the code harder to read, understand, and maintain. It can also make error handling and code reuse more challenging. To mitigate callback hell, alternative approaches like Promises and Async/Await were introduced, which provide cleaner and more structured ways to handle asynchronous operations.

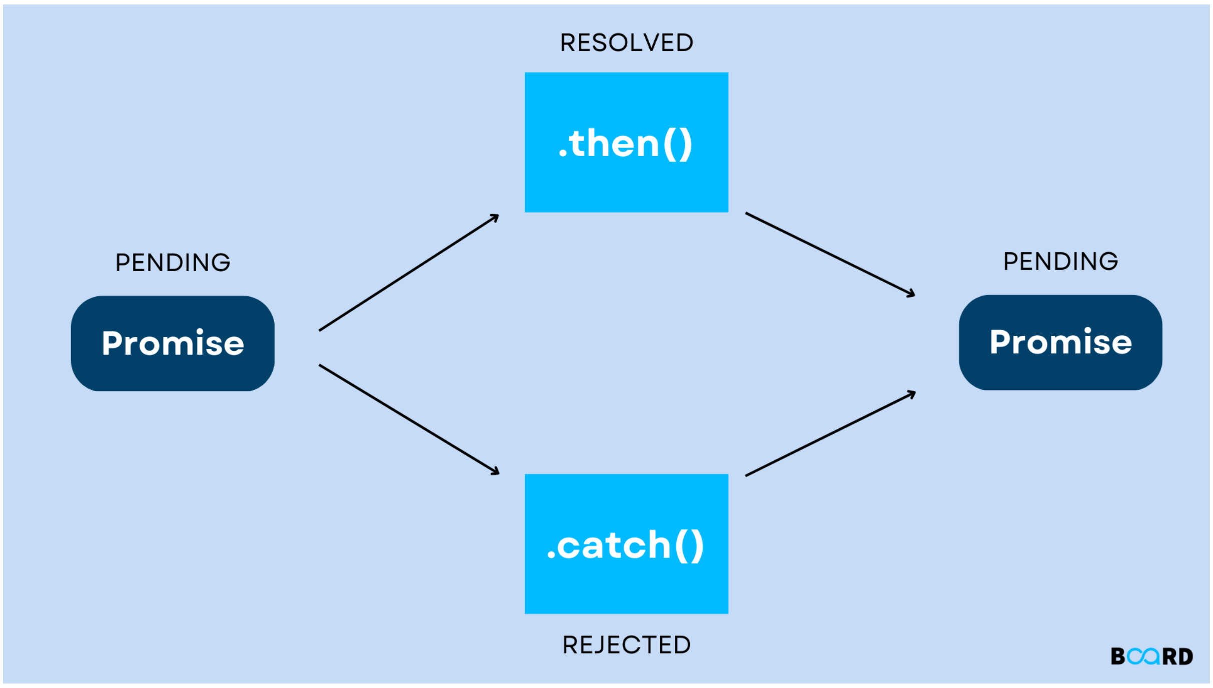
JavaScript Promise

In JavaScript, a promise is a good way to handle **asynchronous** operations. It is used to find out if the asynchronous operation is successfully completed or not.

A promise may have one of three states.

- Pending
- Fulfilled
- Rejected

A promise starts in a pending state. That means the process is not complete. If the operation is successful, the process ends in a fulfilled state. And, if an error occurs, the process ends in a rejected state.



Promise Example

```
function makeSandwich(ingredient1, ingredient2) {
  return new Promise((resolve, reject) => {
    console.log(`Making a sandwich with ${ingredient1} and
${ingredient2}...`);
    setTimeout(() => {
      const sandwich = `🥪(${ingredient1} & ${ingredient2})
`;
      console.log(`Sandwich is ready: ${sandwich}`);
      resolve(sandwich);
    }, 3000);
  });
}

function eatSandwich(sandwich) {
  console.log(`Eating the sandwich: ${sandwich}`);
}

makeSandwich('🍅', '🧀')
  .then(eatSandwich)
  .catch(error => console.error(error));
```

```
console.log('Playing with toys while waiting for the sandwi
ch...');
```

Output

Making a sandwich with 🍅 and 🧀...

Playing with toys while waiting for the sandwich...

Sandwich is ready: 🥪(🍅 & 🧀)

Eating the sandwich: 🥪(🍅 & 🧀)

Promise with reject

```
function makeSandwich(ingredient1, ingredient2) {
    return new Promise((resolve, reject) => {
        if (!ingredient1 || !ingredient2) {
            reject('Missing ingredient!');
            return;
        }
        console.log(`Making a sandwich with ${ingredient1}
and ${ingredient2}...`);
        setTimeout(() => {
            const sandwich = `🥪(${ingredient1} & ${ingredi
ent2})`;
            console.log(`Sandwich is ready: ${sandwich}`);
            resolve(sandwich);
        }, 3000);
    });
}

function eatSandwich(sandwich) {
    console.log(`Eating the sandwich: ${sandwich}`);
}

makeSandwich('🍅', '🧀')
    .then(eatSandwich)
    .catch(error => console.error(`Error: ${error}`));
```

```
console.log('Playing with toys while waiting for the sandwi
ch...');
```

Output

Making a sandwich with 🍅 and 🧀...

Playing with toys while waiting for the sandwich...

Sandwich is ready: 🥪(🍅 & 🧀)

Eating the sandwich: 🥪(🍅 & 🧀)

In this modified version of the code, if one of the ingredients is null, the promise will be rejected and the error callback function passed to .catch will be called with the error message 'Missing ingredient!'. This will log an error message to the console. Otherwise, if both ingredients are present, the code will work as before and make a sandwich with the specified ingredients.

Promise Chaining

```
function makeSandwich(ingredient1, ingredient2) {
    return new Promise((resolve, reject) => {
    console.log(`Making a sandwich with ${ingredient1} and
${ingredient2}...`);
    setTimeout(() => {
    const sandwich = `🥪(${ingredient1} & ${ingredient2})`;
    console.log(`Sandwich is ready: ${sandwich}`);
    resolve(sandwich);
    }, 3000);
    });
    }

    function addCondiments(sandwich) {
    const sandwichWithCondiments = `${sandwich} + 🍅 + 🥒 +
🧂`;
    console.log(`Sandwich with condiments: ${sandwichWithCo
ndiments}`);
    return sandwichWithCondiments;
    }
    function eatSandwich(sandwich) {
    console.log(`Eating the sandwich: ${sandwich}`);
    }

    makeSandwich('🍅', '🧀')
    .then(addCondiments)
    .then(eatSandwich)
    .catch(error => console.error(error));

    console.log('Playing with toys while waiting for the sandwich...');
```

Output

Making a sandwich with 🍅 and 🧀...

Playing with toys while waiting for the sandwich...

Sandwich is ready: 🥪(🍅 & 🧀)

Sandwich with condiments: 🥪(🍅 & 🧀) + 🍅 + 🥒 + 🧂

Eating the sandwich: 🥪(🍅 & 🧀) + 🍅 + 🥒 + 🧂

What is the Job Queue in JavaScript?

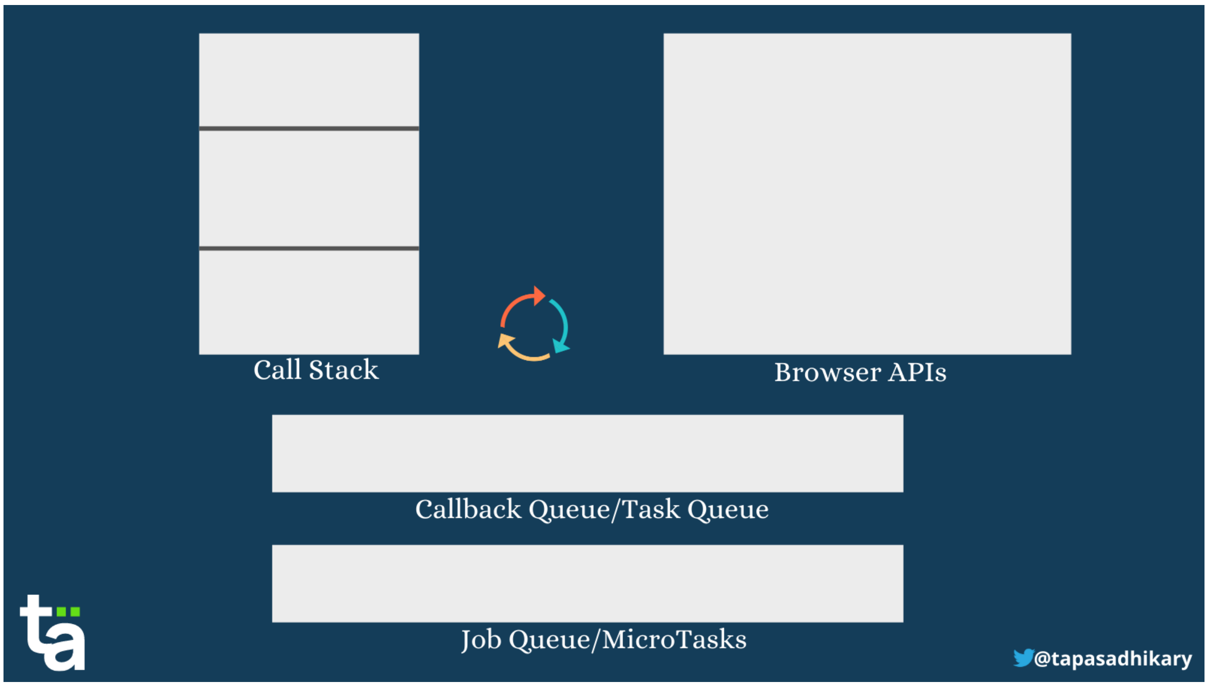
Every time a promise occurs in the code, the executor function gets into the job queue. The event loop works, as usual, to look into the queues but gives priority to the `job queue` items over the `callback queue` items when the `stack` is free.

The item in the callback queue is called a `macro task`, whereas the item in the job queue is called a `micro task`.

So the entire flow goes like this:

- For each loop of the `event loop`, one task is completed out of the `callback queue`.
- Once that task is complete, the event loop visits the `job queue`. It completes all the `micro tasks` in the job queue before it looks into the next thing.
- If both the queues got entries at the same point in time, the `job queue` gets preference over the `callback queue`.

The image below shows the inclusion of the job queue along with other preexisting items.



https://www.freecodecamp.org/news/content/images/2021/09/fourth-flow.gif

Async/await

Async/await is a feature in JavaScript that allows you to work with asynchronous code in a more synchronous-like manner, making it easier to write and understand asynchronous code.

*"async and await make promises easier to write"***async** makes a function return a Promise. **await** makes a function wait for a Promise

```
async function makeAndEatSandwich(ingredient1, ingredient2)
{
    try {
        const sandwich = await makeSandwich(ingredient1, in
gredient2);
        eatSandwich(sandwich);
    } catch (error) {
        console.error(error);
    }
}
```

```
function makeSandwich(ingredient1, ingredient2) {
    return new Promise((resolve, reject) => {
        console.log(`Making a sandwich with ${ingredient1}
and ${ingredient2}...`);
        setTimeout(() => {
            const sandwich = `🥪(${ingredient1} & ${ingredi
ent2})`;
            console.log(`Sandwich is ready: ${sandwich}`);
            resolve(sandwich);
        }, 3000);
    });
}

function eatSandwich(sandwich) {
    console.log(`Eating the sandwich: ${sandwich}`);
}

makeAndEatSandwich('🍅', '🧀');

console.log('Playing with toys while waiting for the sandwi
ch...');
```

In this version, we create a new `async` d `makeAndEatSandwich`. This function uses the` ``await t for the promise returned by the `makeSandwich`` resolved before calling the \`eatSandwich\` f `eatSandwich` The `await` keyword can only be used inside an `async` function. It makes the code look like synchronous code, even though it is still asynchronous under the hood. When we use \`await\`, the execution of the function is paused until the promise is resolved or rejected.

In this example, we use a \`try/catch\` block to handle errors. If there is an error while making the sandwich (e.g., if the promise returned by \`makeSandwich\` is rejected), the error will be caught by the \`catch\` block and logged to the console.

Finally, we call the \`makeAndEatSandwich\` function to make and eat a sandwich with tomato and cheese. We also log a message to the console to indicate that we are doing something else while waiting for the sandwich to be ready. This message will be logged immediately after calling the \`makeAndEatSandwich\` function, because promises are asynchronous and do not block the execution of the rest of the code.

To handle both successful and unsuccessful outcomes

```
async function makeAndEatSandwich(ingredient1, ingredient2)
{
    try {
        const sandwich = await makeSandwich(ingredient1, in
gredient2);
        eatSandwich(sandwich);
    } catch (error) {
        console.error('Oops! Something went wrong:', erro
r);
    }
}

function makeSandwich(ingredient1, ingredient2) {
    return new Promise((resolve, reject) => {
        console.log(`Making a sandwich with ${ingredient1}
and ${ingredient2}...`);
        setTimeout(() => {
            if (Math.random() < 0.5) {
                const sandwich = `🥪(${ingredient1} & ${ing
redient2})`;
                console.log(`Sandwich is ready: ${sandwich}
`);
                resolve(sandwich);
            } else {
                reject(new Error('Oops! Something went wron
g while making the sandwich.'));
            }
        }, 3000);
    });
}

function eatSandwich(sandwich) {
    console.log(`Eating the sandwich: ${sandwich}`);
}

makeAndEatSandwich('🍅', '🧀');
```

```
console.log('Playing with toys while waiting for the sandwi
ch...');
```

In this updated code, the makeSandwich function randomly decides whether to resolve or reject the Promise. If it resolves, the makeAndEatSandwich function proceeds to eat the sandwich. If it rejects, the catch block in makeAndEatSandwich handles the error and logs it to the console.

Alternatives for handling asynchronous operations in JavaScript:

1. **Callbacks**: Callbacks are the traditional way to handle asynchronous code. You pass a function (the callback) as an argument to an asynchronous operation, and it gets executed when the operation completes. However, nested callbacks can lead to callback hell and make code harder to read and maintain.
2. **Async/await**: Introduced in ECMAScript 2017 (ES8), `async` functions allow you to write asynchronous code in a more synchronous style. You use the `await` keyword inside an `async` function to pause execution until a promise resolves. This approach is more readable than callbacks and avoids callback hell.
3. **Generators**: Although less common, generators can be used for asynchronous control flow. You create a generator function using the `function*` syntax and yield promises. You then iterate over the generator using `next()` and handle the resolved values.
4. **Observables (RxJS)**: Observables are part of reactive programming libraries like RxJS. They provide a powerful way to handle asynchronous events, streams, and data flows. Observables are especially useful for handling real-time data and complex scenarios.