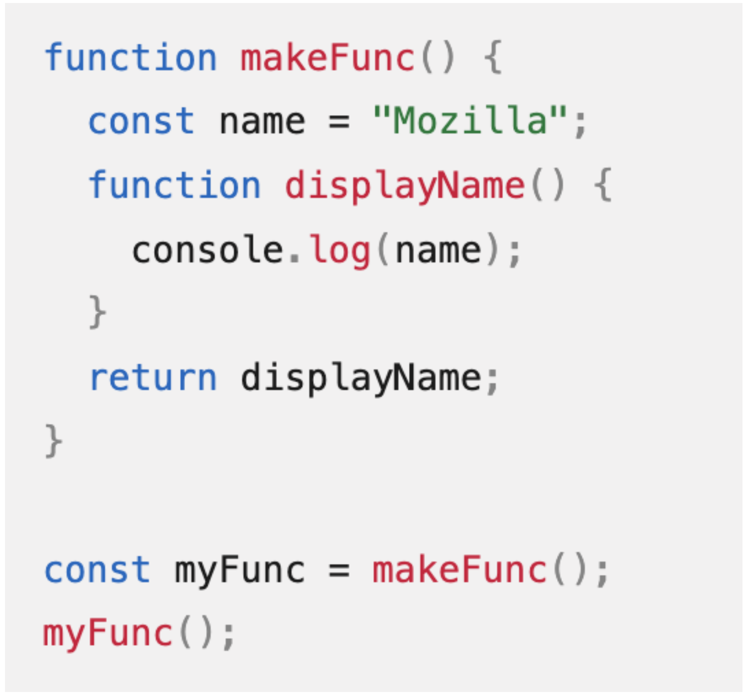
Advance JavaScript

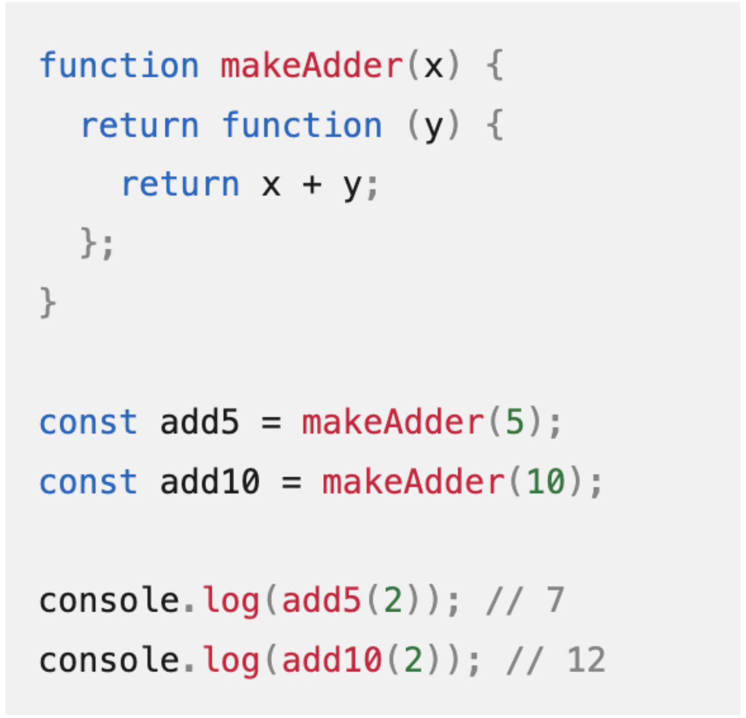
IF3110 – Web-based Application Development
School of Electical Engineering and Informatics
Institut Teknologi Bandung

Closure

A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives a function access to its outer scope. In JavaScript, closures are created every time a function is created, at function creation time.



```
function makeFunc() {
  const name = "Mozilla";
  function displayName() {
    console.log(name);
  }
  return displayName;
}

const myFunc = makeFunc();
myFunc();
```



```
function makeAdder(x) {
  return function (y) {
    return x + y;
  };
}

const add5 = makeAdder(5);
const add10 = makeAdder(10);

console.log(add5(2)); // 7
console.log(add10(2)); // 12
```

Memory Management

Regardless of the programming language, the memory life cycle is pretty much always the same:

1. Allocate the memory you need
2. Use the allocated memory (read, write)
3. Release the allocated memory when it is not needed anymore

JavaScript automatically allocates memory when objects are created and frees it when they are not used anymore (garbage collection)

Memory Management: Allocation

Value initialization

```
const n = 123; // allocates memory for a number
const s = "azerty"; // allocates memory for a string

const o = {
  a: 1,
  b: null,
}; // allocates memory for an object and contained values

// (like object) allocates memory for the array and
// contained values
const a = [1, null, "abra"];

function f(a) {
  return a + 2;
} // allocates a function (which is a callable object)

// function expressions also allocate an object
someElement.addEventListener(
  "click",
  () => {
    someElement.style.backgroundColor = "blue";
  },
  false,
);
```

Function Call

```
const d = new Date(); // allocates a Date object

const e = document.createElement("div"); // allocates a DOM element
```

Memory Management: Using Values

Using values basically means reading and writing in allocated memory. This can be done by reading or writing the value of a variable or an object property or even passing an argument to a function.

Memory Management: Garbage collection

The majority of memory management issues occur at this phase. The most difficult aspect of this stage is determining when the allocated memory is no longer needed.

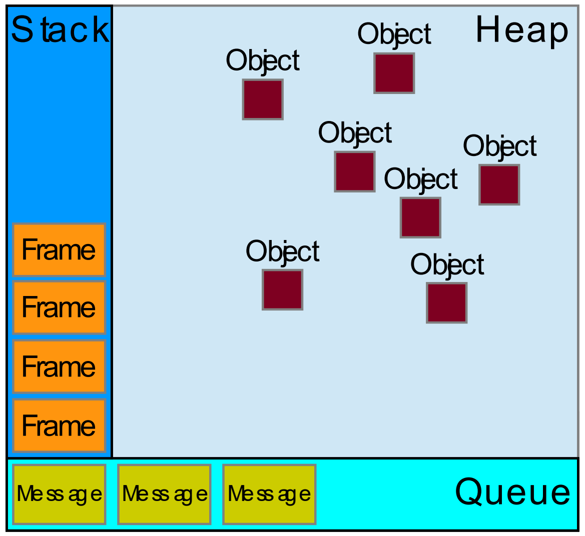
The main concept that garbage collection algorithms rely on is the concept of reference. Within the context of memory management, an object is said to reference another object if the former has access to the latter (either implicitly or explicitly).

Reference-counting Algorithm -> determining whether or not an object is still needed to determining if an object still has any other objects referencing it (not used anymore in modern JavaScript engine)

Mark-and-sweep Algorithm -> reduces the definition of "an object is no longer needed" to "an object is unreachable" (all modern engines ship a mark-and-sweep garbage collector)

Concurrency model and Event Loop

JavaScript has a runtime model based on an **event loop**, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks. This model is quite different from models in other languages like C and Java.



JavaScript runtime concepts (only a theoretical model)

Modern JavaScript engines implement and heavily optimize this.

JavaScript program is *single-threaded*. A thread is a sequence of instructions that a program follows. Because the program consists of a single thread, it can only do one thing at a time: so if it is waiting for our long-running synchronous call to return, it can't do anything else.

Runtime Concept: Stack

Function calls form a stack of frames.

```javascript
function foo(b) {
  const a = 10;
  return a + b + 11;
}

function bar(x) {
  const y = 3;
  return foo(x * y);
}

const baz = bar(7); // assigns 42 to baz
```

Order of operations:

1. When calling ***bar***, a first frame is created containing references to bar's arguments and local variables.

2. When bar calls ***foo***, a second frame is created and pushed on top of the first one, containing references to foo's arguments and local variables.

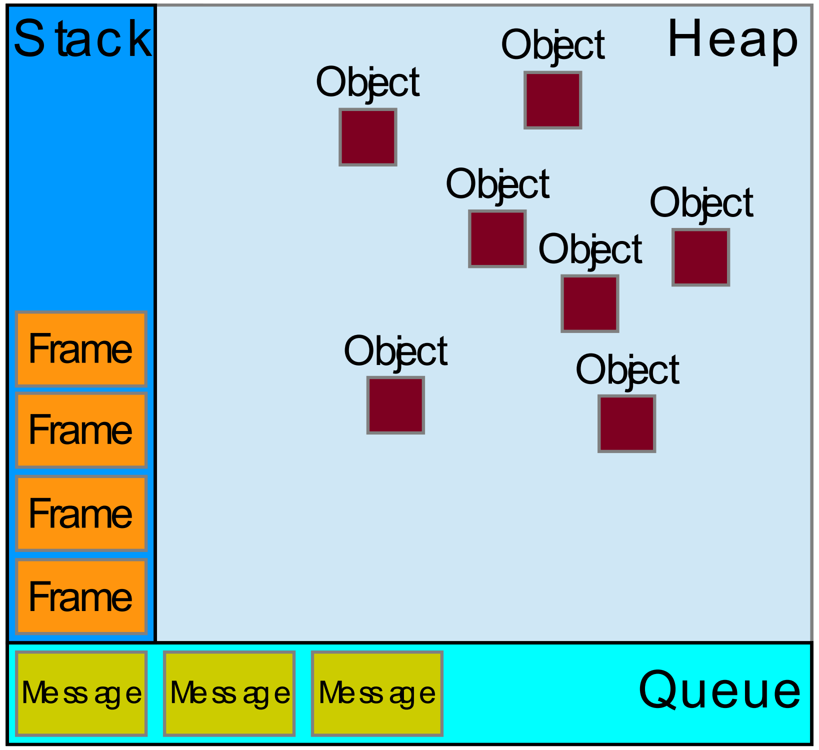
3. When ***foo*** returns, the top frame element is popped out of the stack (leaving only bar's call frame).

4. When ***bar*** returns, the stack is empty.

Note that the arguments and local variables may continue to exist, as they are stored outside the stack

Runtime Concept: Heap

Objects are allocated in a heap which is just a name to denote a large (mostly unstructured) region of memory.



Runtime Concept: Queue

A JavaScript runtime uses a message queue, which is a list of messages to be processed. Each message has an associated function that gets called to handle the message.

At some point during the event loop, the runtime starts handling the messages on the queue, starting with the oldest one. To do so, the message is removed from the queue and its corresponding function is called with the message as an input parameter. As always, calling a function creates a new stack frame for that function's use.

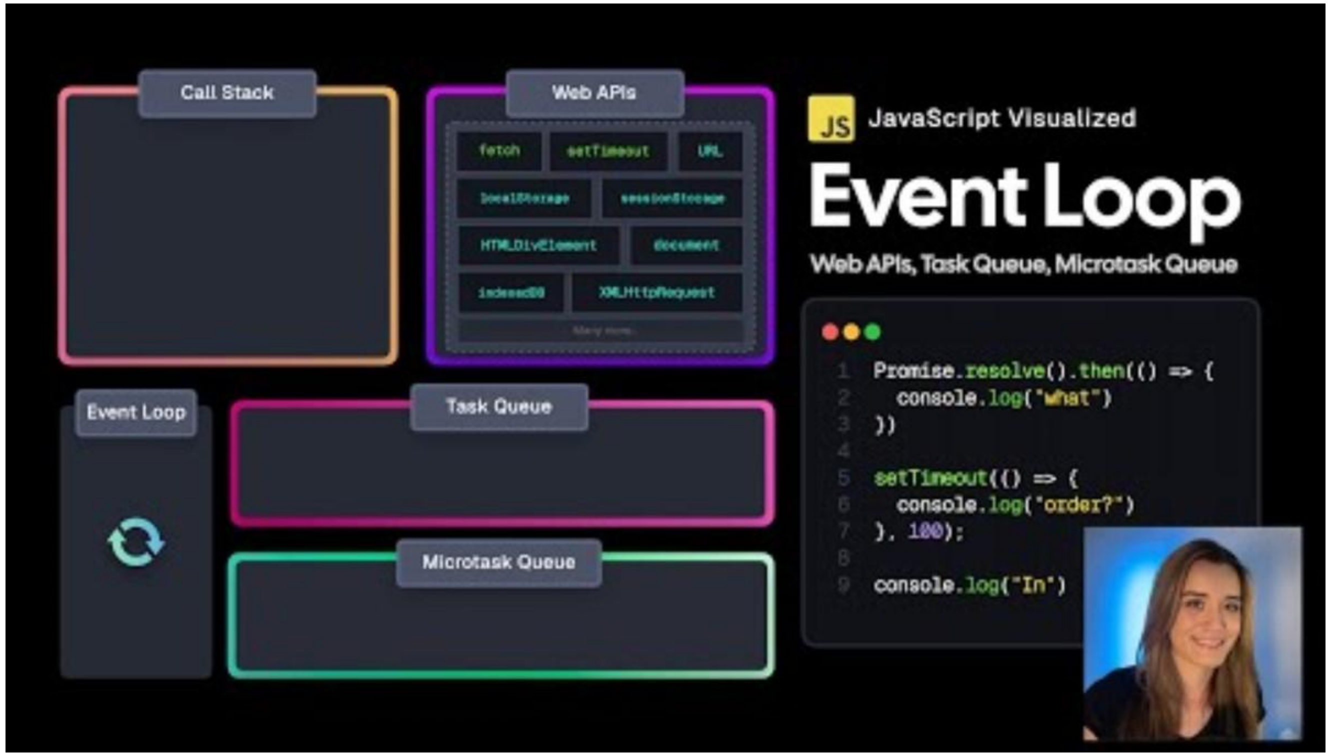
The processing of functions continues until the stack is once again empty. Then, the event loop will process the next message in the queue (if there is one).

The Event Loop

The event loop got its name because of how it's usually implemented, which usually resembles:

```
while (queue.waitForMessage()) {
  queue.processNextMessage();
}
```

A very interesting property of the event loop model is that JavaScript, unlike a lot of other languages, never blocks.
Legacy exceptions exist like alert or synchronous XHR, but it is considered good practice to avoid them



```
Promise.resolve().then(() => {
  console.log("what")
})

setTimeout(() => {
  console.log("order?")
}, 100);

console.log("In")
```

Good Visualization of the Event Loop:
https://www.youtube.com/watch?v=eiC58R16hb8

Asynchronous JavaScript

When multiple related things happen without any being dependent on the completion of previous happenings, they are asynchronous.

```
const MAX_PRIME = 1000000;

function isPrime(n) {
  for (let i = 2; i <= Math.sqrt(n); i++) {
    if (n % i === 0) {
      return false;
    }
  }
  return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generatePrimes(quota) {
  const primes = [];
  while (primes.length < quota) {
    const candidate = random(MAX_PRIME);
    if (isPrime(candidate)) {
      primes.push(candidate);
    }
  }
  return primes;
}
```

We have synchronous program that a very inefficient algorithm to generate multiple large prime numbers.

What if the synchronous function takes a long time?

A downside of the JavaScript runtime model is that if a function takes too long to complete, the web application is unable to process user interactions like click or scroll. The browser mitigates this with the "a script is taking too long to run" dialog.

Asynchronous JavaScript: Event Handler

Event handlers are really a form of asynchronous programming: you provide a function (the event handler) that will be called, not right away, but whenever the event happens. If "the event" is "the asynchronous operation has completed", then that event could be used to notify the caller about the result of an asynchronous function call.

Example using asynchronous API: XMLHttpRequest

```javascript
const log = document.querySelector(".event-log");

document.querySelector("#xhr").addEventListener("click", () => {
  log.textContent = "";

  const xhr = new XMLHttpRequest();

  xhr.addEventListener("loadend", () => {
    log.textContent = `${log.textContent}Finished with status: ${xhr.status}`;
  });

  xhr.open(
    "GET",
    "https://raw.githubusercontent.com/mdn/content/main/files/en-
us/_wikihistory.json",
  );
  xhr.send();
  log.textContent = `${log.textContent}Started XHR request\n`;
});
```

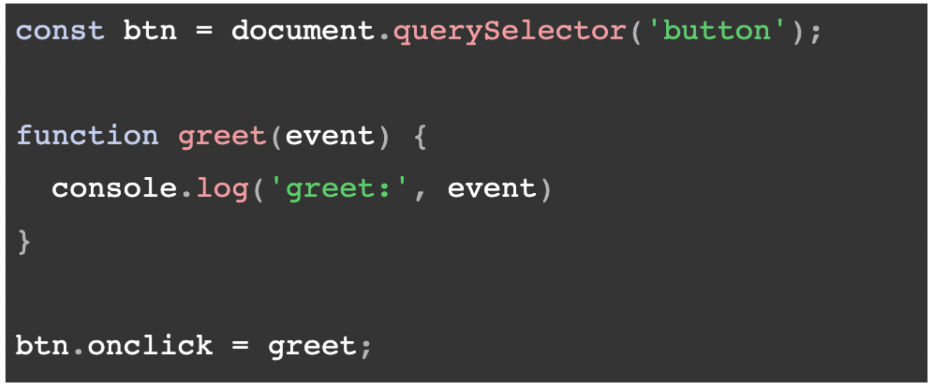
About Event Listeners in JavaScript

- Event Handler concept

Event Handler is function that handles when particular Event happened at specified target. Object representing the event is passed as the first argument to the event handler. This event object either implements or is derived from the Event interface.

Used not only on UI, but also in workers, websocket, etc.

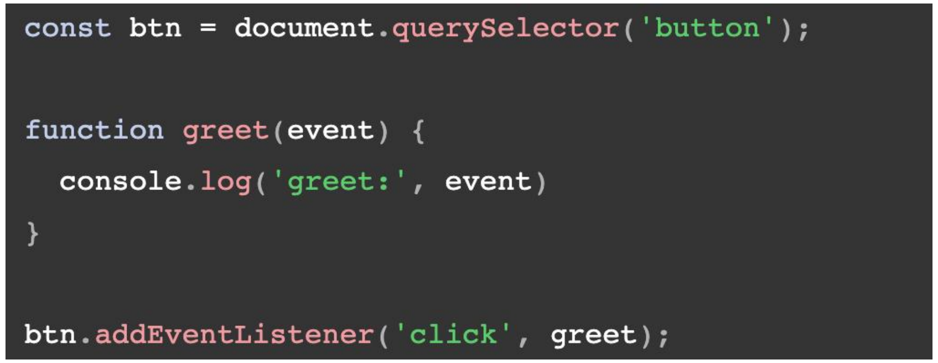
- Two approaches:
  - onevent: simplicity and one to one.



```
const btn = document.querySelector('button');

function greet(event) {
  console.log('greet:', event)
}

btn.onclick = greet;
```

  - addEventListener and removeEventListener:

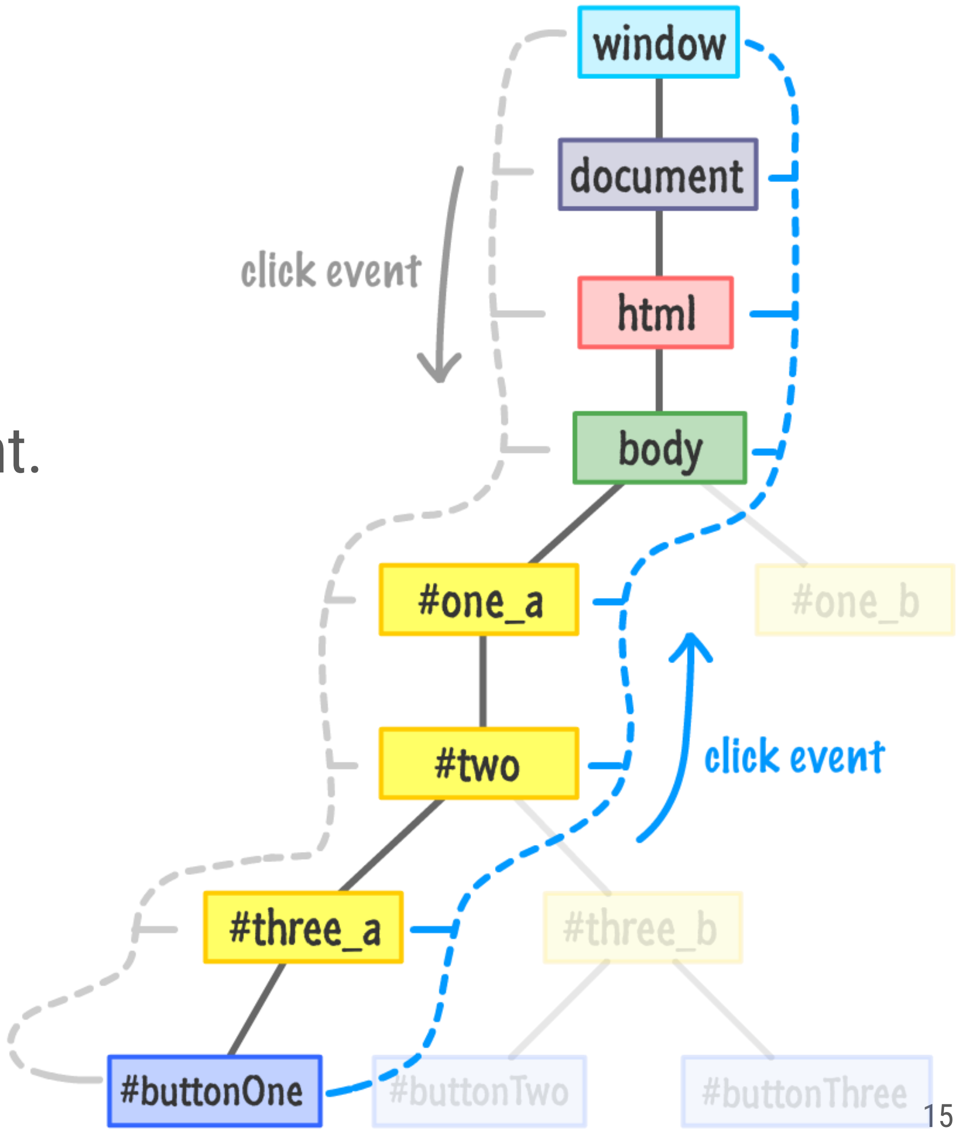
  Very Flexible and Specific, Allows multiple event listener attached to a target. Also enable Event Bubbling/Capture phase to be specified (for DOM event)



```
const btn = document.querySelector('button');

function greet(event) {
  console.log('greet:', event)
}

btn.addEventListener('click', greet);
```

Event Bubble/Capture

- The standard DOM Events describes 3 phases of event propagation:
  - Capturing phase – the event goes down to the element.
  - Target phase – the event reached the target element.
  - Bubbling phase – the event bubbles up from the element.
- Event.target is the deepest nested element being the target.
- ”this” refer to the current element.



Asynchronous JavaScript: Callback

A callback is just a function that's passed into another function, with the expectation that the callback will be called at the appropriate time.

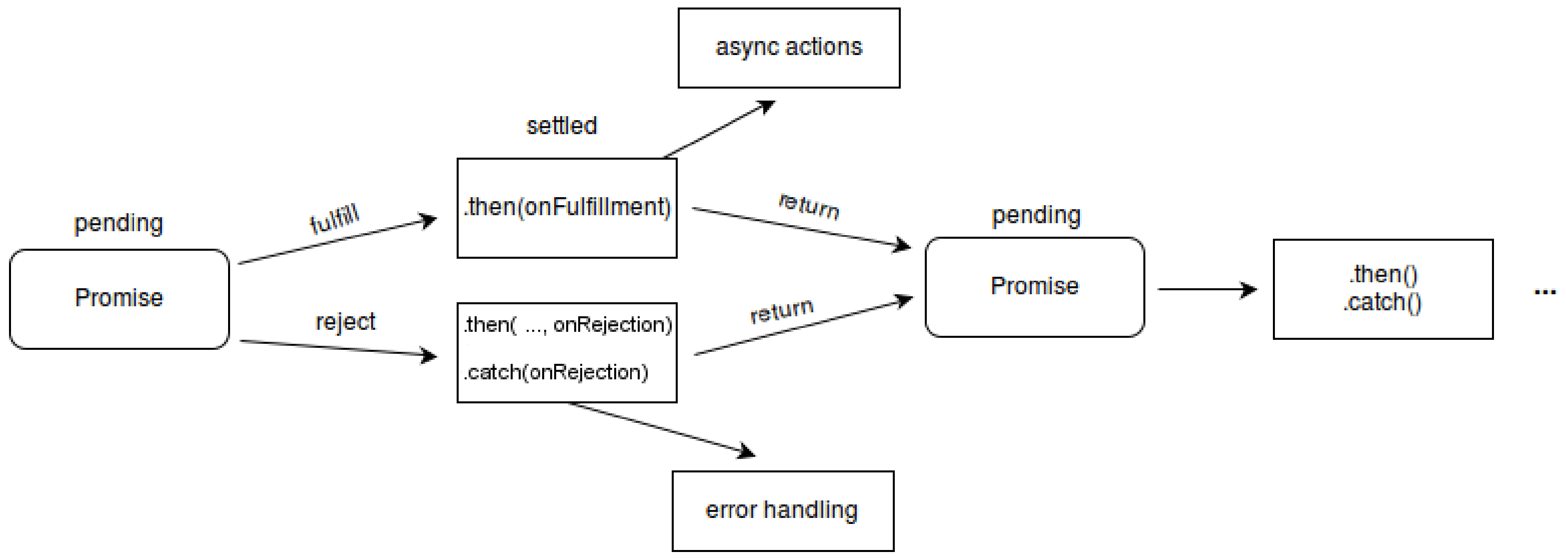
```javascript
function doStep1(init, callback) {
  const result = init + 1;
  callback(result);
}

function doStep2(init, callback) {
  const result = init + 2;
  callback(result);
}

function doStep3(init, callback) {
  const result = init + 3;
  callback(result);
}

function doOperation() {
  doStep1(0, (result1) => {
    doStep2(result1, (result2) => {
      doStep3(result2, (result3) => {
        console.log(`result: ${result3}`);
      });
    });
  });
}

doOperation();
```

However, callback-based code can get hard to understand when the callback itself has to call functions that accept a callback. This is a common situation if you need to perform some operation that breaks down into a series of asynchronous functions.

This is sometimes called "callback hell" or the "pyramid of doom".

Asynchronous JavaScript: Promise

The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason



pending: initial state, neither fulfilled nor rejected.

fulfilled: meaning that the operation was completed successfully.

rejected: meaning that the operation failed.

Asynchronous JavaScript: Promise creation and chaining

Example below create and return new Promise Object. Then we handle or reject using then interface.

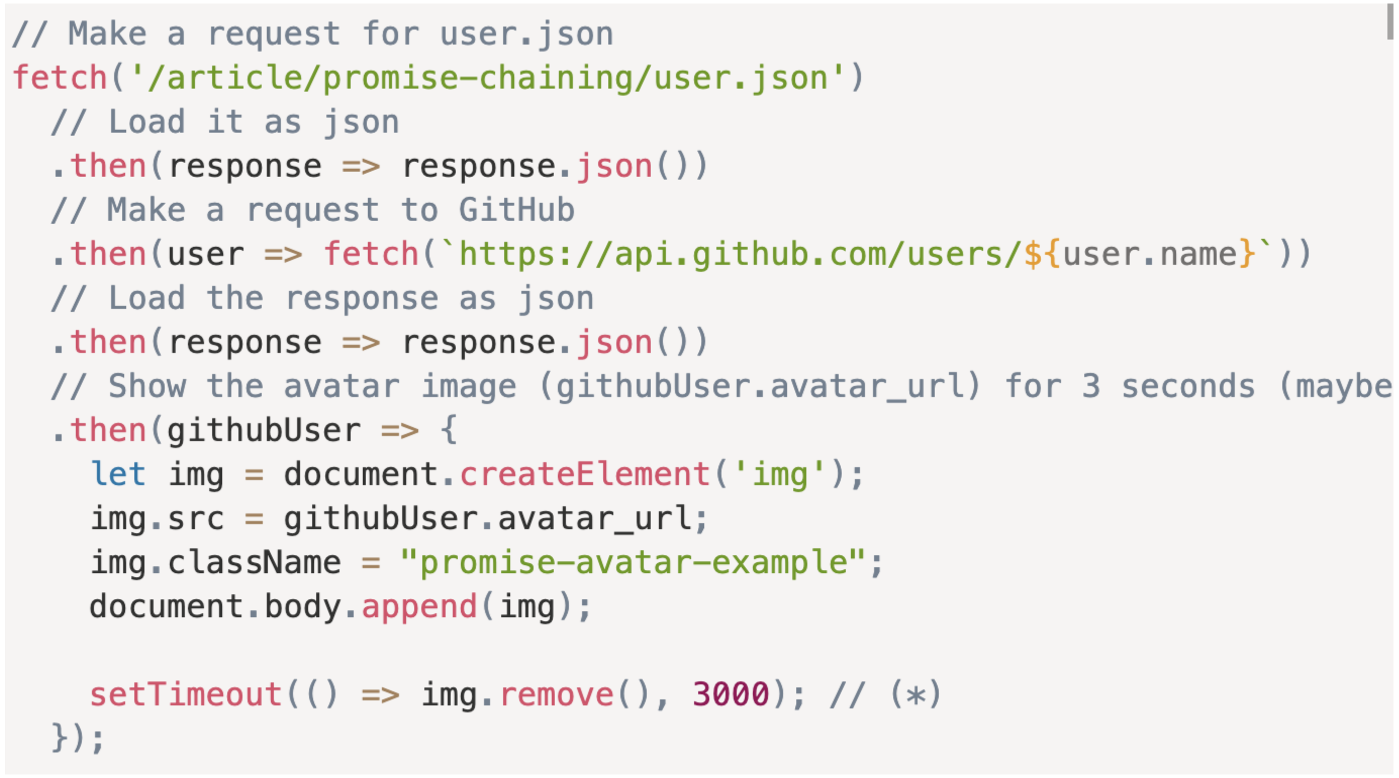
```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("foo");
  }, 300);
});

myPromise
  .then(handleFulfilledA, handleRejectedA)
  .then(handleFulfilledB, handleRejectedB)
  .then(handleFulfilledC, handleRejectedC);
```

The promise chain is what you need when your operation consists of several asynchronous functions, and you need each one to complete before starting the next one.

This special then() function returns another promise p. Depends on the return of value of the handler which may:

- returns a value: p gets **fulfilled** with the returned value as its value.
- doesn't return anything: p gets **fulfilled** with undefined as its value.
- throws an error: p gets **rejected** with the thrown error as its value.
- returns another pending promise: p is **pending** and becomes fulfilled/rejected with that promise's value as its value immediately after that promise becomes fulfilled/rejected.

Asynchronous JavaScript: Fetch example



```
// Make a request for user.json
fetch('/article/promise-chaining/user.json')
  // Load it as json
  .then(response => response.json())
  // Make a request to GitHub
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  // Load the response as json
  .then(response => response.json())
  // Show the avatar image (githubUser.avatar_url) for 3 seconds (maybe
  .then(githubUser => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => img.remove(), 3000); // (*)
  });
```

Fetch is the modern replacement for XMLHttpRequest: unlike XMLHttpRequest (XHR), which uses callbacks, Fetch is promise-based.

The fetch() function returns a Promise which is fulfilled with a Response object representing the server's response. You can then check the request status and extract the body of the response in various formats, including text and JSON, by calling the appropriate method on the response.

Asynchronous JavaScript: Async Await

The async keyword gives you a simpler way to work with asynchronous promise-based code. Adding async at the start of a function makes it an async function:

```
async function fetchProducts() {
  try {
    // after this line, our function will wait for the `fetch()` call to be settled
    // the `fetch()` call will either return a Response or throw an error
    const response = await fetch(
      "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-
store/products.json",
    );
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    // after this line, our function will wait for the `response.json()` call to be
settled
    // the `response.json()` call will either return the parsed JSON object or throw
an error
    const data = await response.json();
    console.log(data[0].name);
  } catch (error) {
    console.error(`Could not get products: ${error}`);
  }
}

fetchProducts();
```

Inside an async function, you can use the await keyword before a call to a function that returns a promise.

The async and await keywords make it easier to build an operation from a series of consecutive asynchronous function calls, avoiding the need to create explicit promise chains, and allowing you to write code that looks just like synchronous code (a cleaner style).