Message Queue and Event Loop

JavaScript is a single-threaded, non-blocking, asynchronous programming language. It achieves this through the **event loop**, which allows it to handle multiple tasks without blocking the main thread, such as user interactions, file I/O, and network requests.

**1. Call Stack**

The call stack is a data structure that keeps track of the execution context: the currently executing function and its ancestor functions. When a function is called, it is added to the top of the stack. When the function execution is complete, it is removed from the stack.

**2. Message Queue**

The message queue is where tasks (like events, timers, or callbacks) wait to be executed. These tasks are queued as messages in the message queue, and the event loop picks them up when the call stack is empty.

**3. Event Loop**

The event loop is a mechanism that continuously checks the call stack to see if it's empty. If the stack is empty, it takes the first message from the message queue and pushes its associated callback onto the call stack for execution.

**4. Web APIs**

Certain asynchronous functions (e.g., setTimeout, fetch) are not part of JavaScript itself but provided by the browser or Node.js environment. These APIs handle the asynchronous tasks and place their callbacks in the message queue when they are ready.

**5. Microtasks**

Microtasks (e.g., Promise.then, MutationObserver) have higher priority than normal tasks in the message queue. They are executed before moving to the next message in the queue.

Example: Asynchronous Task Execution

console.log("Start");

setTimeout(() => {

console.log("Timeout");

}, 0);

Promise.resolve().then(() => {

console.log("Promise");

});

**Step-by-Step Execution**

1. **Initial Call Stack:**
   * console.log("Start") is executed and logs Start.
   * Call stack is empty.
2. **setTimeout:**
   * The setTimeout callback is sent to the Web APIs.
   * After 0 ms (minimum threshold), the callback (console.log("Timeout")) is added to the **Message Queue**.
3. **Promise:**
   * Promise.resolve().then is a microtask. Its callback (console.log("Promise")) is added to the **Microtask Queue**.
4. **console.log("End"):**
   * Logs End.
   * Call stack is empty.
5. **Microtask Queue:**
   * The microtask queue is checked, and console.log("Promise") is executed.
6. **Message Queue:**
   * The event loop picks the setTimeout callback (console.log("Timeout")) from the message queue and executes it.

**Visualization of Event Loop**

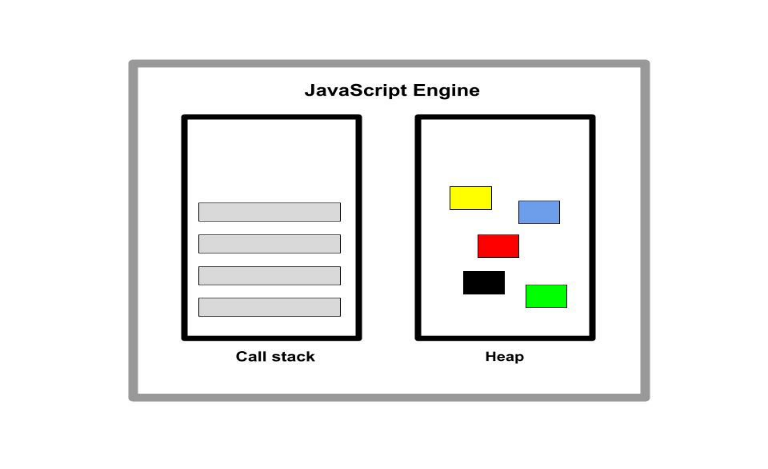
1. **Call Stack:**
   * Executes synchronous code (e.g., console.log).
2. **Microtask Queue:**
   * Processes all microtasks before handling the next task in the message queue.
3. **Message Queue:**
   * Processes tasks like setTimeout after the call stack is empty and all microtasks are handled.

Javascript Engines

**JavaScript engines are interpreters that parse and execute JavaScript code. Modern JavaScript engines use just-in-time (JIT) compilation to convert JavaScript code into machine code that can be executed by a computer's processor. A JavaScript engine is typically developed and used in web browsers to run client-side code but can also be used in server-side environments like Node.js.**

**Popular JavaScript Engines:**

1. **V8 Engine** (Google Chrome, Node.js): Developed by Google, written in C++.
2. **SpiderMonkey** (Mozilla Firefox): Mozilla's JavaScript engine, the first-ever JavaScript engine.
3. **JavaScriptCore (JSC)** (Safari): Apple's engine, also known as Nitro.
4. **Chakra** (Microsoft Edge Legacy): Microsoft's engine for the Edge browser before the switch to Chromium.

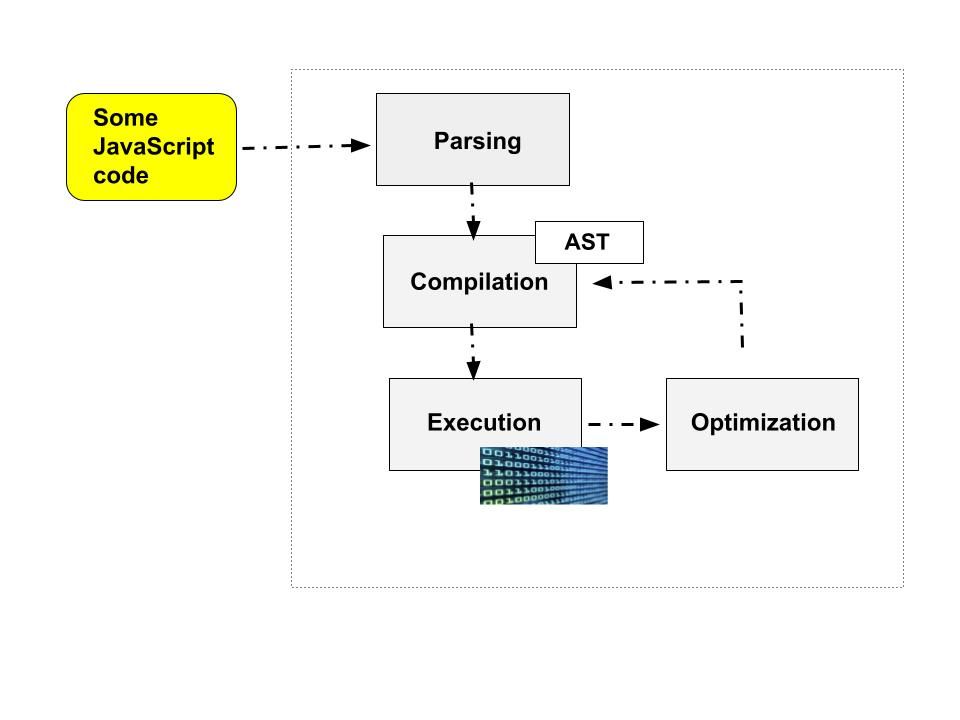


Any JavaScript engine typically contains a call stack and a heap. The call stack is where the code is executed. The heap is an unstructured memory pool that stores all the objects needed for the application.

Since the computer’s processor only understands binary, 0’s and 1’s, the code has to be translated to 0’s and 1’s.

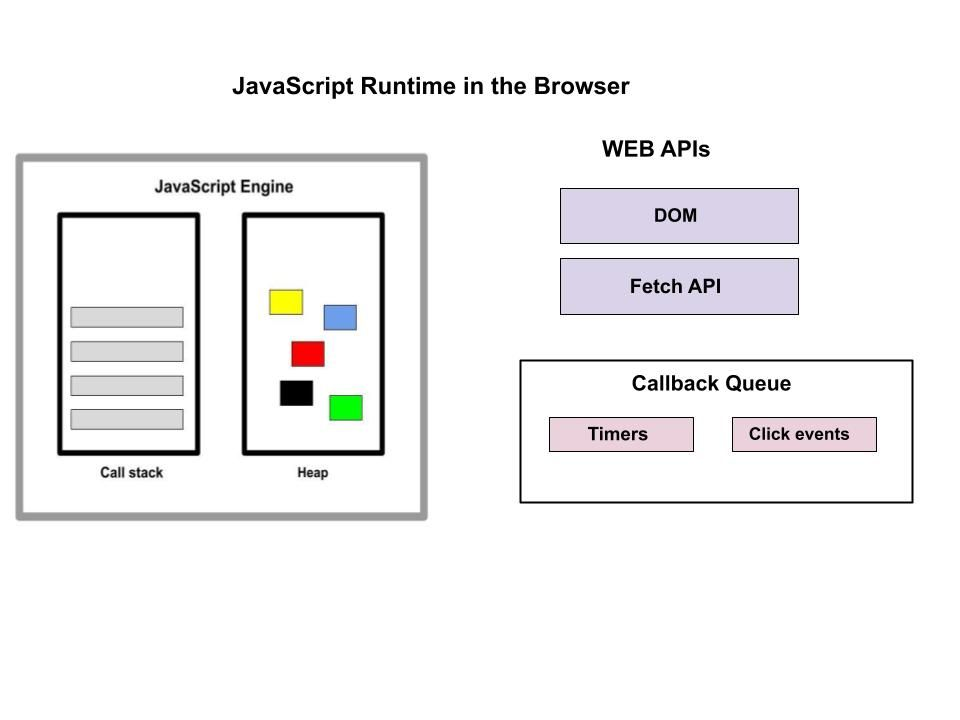
When a code snippet passes into the engine, the code is initially parsed, that is read. The code is subsequently parsed to a data structure called the abstract syntax tree (AST). The resulting tree is then used to create machine codes.

Execution happens in the JavaScript engine call stack using the execution context. This is the environment where JavaScript code is executed.



Think of the JavaScript runtime as the house that encompasses all the components needed to run JavaScript. This house comprises the JavaScript engine, Web APIs, and the callback queue.

Web APIs are functionalities that are provided to the engine but are not part of the JavaScript language. They are accessible to the engine through the browser and help access data or enhance browser functionality. Examples are the Document Object Model (DOM) and Fetch APIs.



The callback queue includes callback functions that are ready to be executed. The callback queue ensures that callbacks are executed in the First-In-First-Out (FIFO) method and they get passed into the stack when it’s empty.

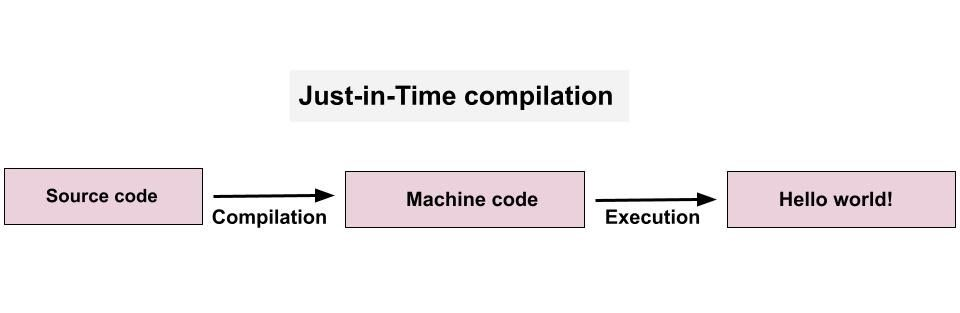
Node.js provides a server-side runtime environment for executing JavaScript outside the browser. Because it executes JavaScript outside the browser, it does not have access to the web APIs. Instead, the Node.js runtime environment replaces it with something called C++ bindings and the thread pool.

**JavaScript Optimization Strategies**

**Just-in-Time compilation**

JavaScript used to be an interpreted language, but interpreted languages are slower compared to compiled languages.

In order to optimize the performance of web applications, JavaScript combines both compilation and interpretation. This is called Just-in-Time compilation. This method compiles the entire code into machine code all at once and executes it.



Just-in-Time compilation involves the same two processes as regular compilation, but here the machine code isn’t written into a binary file. The code is also executed right away after compilation.

To fully optimize JavaScript code, the engine first creates an unoptimized version of the machine code so it can start executing immediately. While that is ongoing, the code is being re-optimized and recompiled in the background of the currently running program execution. This is done multiple times to produce the final, most optimized version.

The process of parsing, compilation, and execution happens in some special thread in the engine that can’t be accessed from the code.

Bitwise Operators, Typed Arrays, and Array Buffers

**Bitwise Operators, Typed Arrays, and Array Buffers** are essential tools in JavaScript, especially when dealing with low-level data manipulation, binary data processing

Bitwise Operators

|  |  |
| --- | --- |
| & | Bitwise AND |

|  |  |  |
| --- | --- | --- |
| ` |  | Bitwise OR |

|  |  |
| --- | --- |
| ^ | Bitwise XOR |

|  |  |
| --- | --- |
| ~ | Bitwise NOT (inverts all bits) |

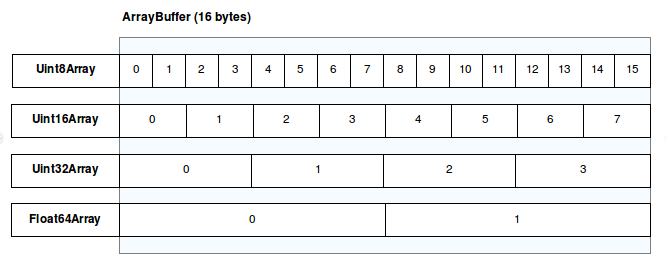
|  |  |  |
| --- | --- | --- |
| << | Left Shift (shifts bits to the left) | 5 << 1 → 10 |

|  |  |  |
| --- | --- | --- |
| >> | Right Shift (shifts bits to the right) | 5 >> 1 → 2 |

|  |  |  |
| --- | --- | --- |
| >>> | Zero-fill Right Shift (shifts right and fills zeros) | -5 >>> 1 → 2147483645 |
| Typed Arrays |  |  |

JavaScript typed arrays are array-like objects that provide a mechanism for reading and writing raw binary data in memory buffers.

JavaScript typed arrays split the implementation into *buffers* and *views*. A buffer is an object representing a chunk of data; it has no format to speak of, and offers no mechanism for accessing its contents. In order to access the memory contained in a buffer, you need to use a view. A view provides a *context* — that is, a data type, starting offset, and number of elements.



DOM and Layout Trees

In JavaScript and web development, the **DOM (Document Object Model)** and **Layout Tree** are fundamental concepts for understanding how browsers interpret and render web pages.

**1. DOM (Document Object Model):**

* The **DOM** is a tree-like representation of the structure and content of an HTML document.
* The DOM is created by the browser's **HTML parser** when the page loads.

**Key Features of the DOM:**

* **Tree Structure:** The document is represented as a hierarchical structure of nodes.
  + **Parent-Child Relationship:** Elements nested within others are children of their parent elements.
  + **Sibling Relationship:** Elements at the same level are siblings.
* **Live Updating:** Changes made to the DOM using JavaScript are immediately reflected in the browser.

**2. Layout Tree:**

* The **Layout Tree** is a visual representation of how elements in the DOM will be rendered on the screen.
* While the DOM includes all elements in the HTML, the Layout Tree only includes elements that are visible and have a visual representation (e.g., display: none elements are excluded).
* The Layout Tree is generated after the browser combines the **DOM** with the **CSSOM** (CSS Object Model), which is the browser's interpretation of all styles applied to elements.

**Key Features of the Layout Tree:**

* **Includes Dimensions and Positions:** Each node in the Layout Tree contains information about the size and position of the element on the page.
* **Basis for Rendering:** The Layout Tree is used by the browser to calculate the exact positions of elements during the rendering process.

**How the DOM and Layout Tree Interact:**

1. **HTML Parsing:** The browser parses the HTML and creates the DOM.
2. **CSS Parsing:** The browser parses the CSS to create the CSSOM.
3. **Layout Calculation:** The browser combines the DOM and CSSOM to create the Layout Tree.
4. **Rendering:** The Layout Tree is used to paint the elements on the screen.

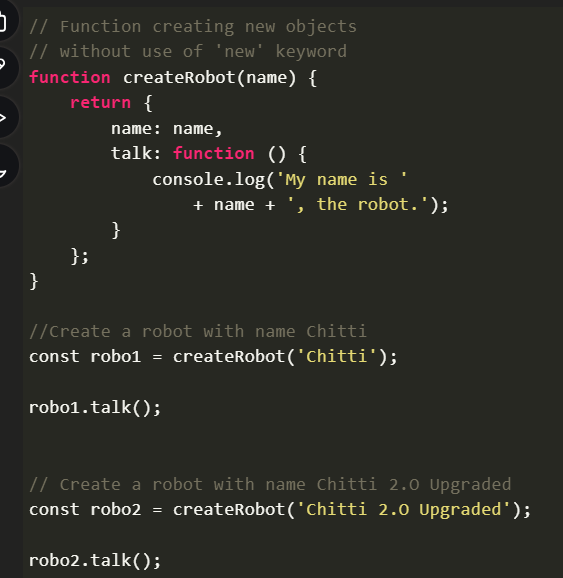
Factories and Classes

JavaScript provides multiple ways to implement object-oriented programming principles, and two common approaches are using **factories** and **classes**

In JavaScript, a factory function is a function that returns an object. It is a way of creating and returning objects in a more controlled and customizable manner. Factory functions are a form of design pattern that enables the creation of objects with specific properties and behaviors.

**Why it is useful?**

If we have complex logic, and we have to create multiple objects again and again that have the same logic, we can write the logic once in a function and use that function as a factory to create our objects. It’s the same as a real-world factory producing products.



**Advantages of Factories:**

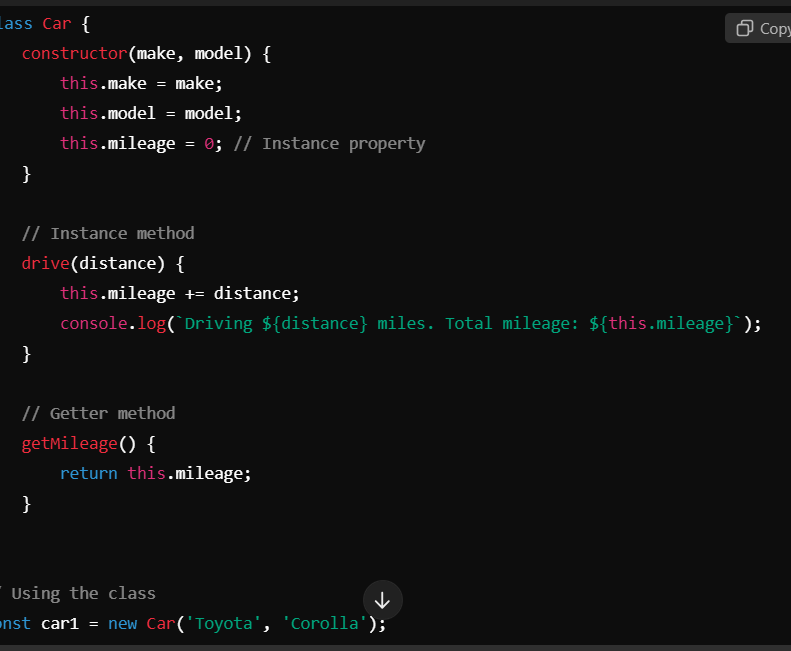
* Encapsulation: Private variables are possible through closures.
* Simplicity: No new keyword or this binding complexities.
* Flexibility: You can easily compose objects from multiple factories.

**Disadvantages of Factories:**

* Performance: Factory functions do not take advantage of prototype inheritance, which can make object creation less memory efficient.
* No inheritance out-of-the-box.

A **class** is a blueprint for creating objects with predefined properties and methods. Introduced in ES6, classes provide a more structured way to define objects in JavaScript while maintaining the prototypal inheritance model under the hood.

Methods are automatically added to the prototype (shared among all instances).



**Advantages of Classes:**

* Better performance due to prototype-based inheritance.
* Readable syntax that aligns with traditional OOP languages.
* Built-in inheritance via extends and super.

**Disadvantages of Classes:**

* Less flexible compared to factories.
* this keyword can sometimes be confusing for beginners.

**When to Use Factory or Class**

* **Factories**:
  + When you need simple, composable objects.
  + When encapsulation and private properties are important.
  + In functional programming scenarios.
* **Classes**:
  + When working with inheritance and a clear object hierarchy.
  + When optimizing for performance.
  + When you need to align with traditional OOP concepts.

this, call, apply, and bind

In JavaScript, this, call, apply, and bind are all related to how the context (this) of a function is determined or manipulated.

**this**

The this keyword in JavaScript refers to the object that is currently executing the function. Its value depends on how the function is called:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**A Simple Analogy**

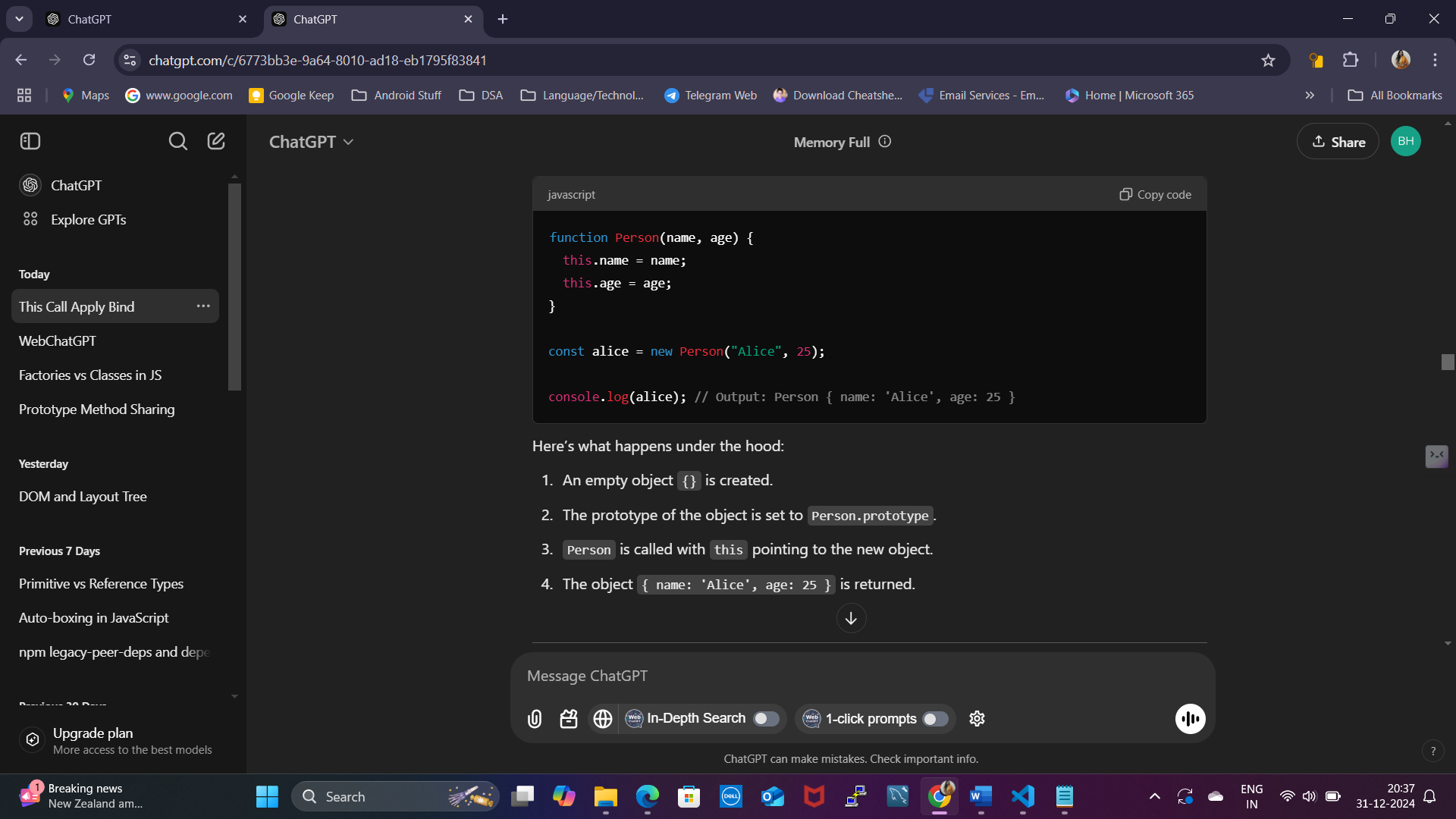
* **this:** "I am the one speaking."
* **call:** "Speak as if you're someone else (immediately)."
* **apply:** "Speak as if you're someone else (immediately), and here's a list of things to say."
* **bind:** "Let me give you a script that you can use later to speak as me."

new, Constructor, instanceof, and Instances

1. New

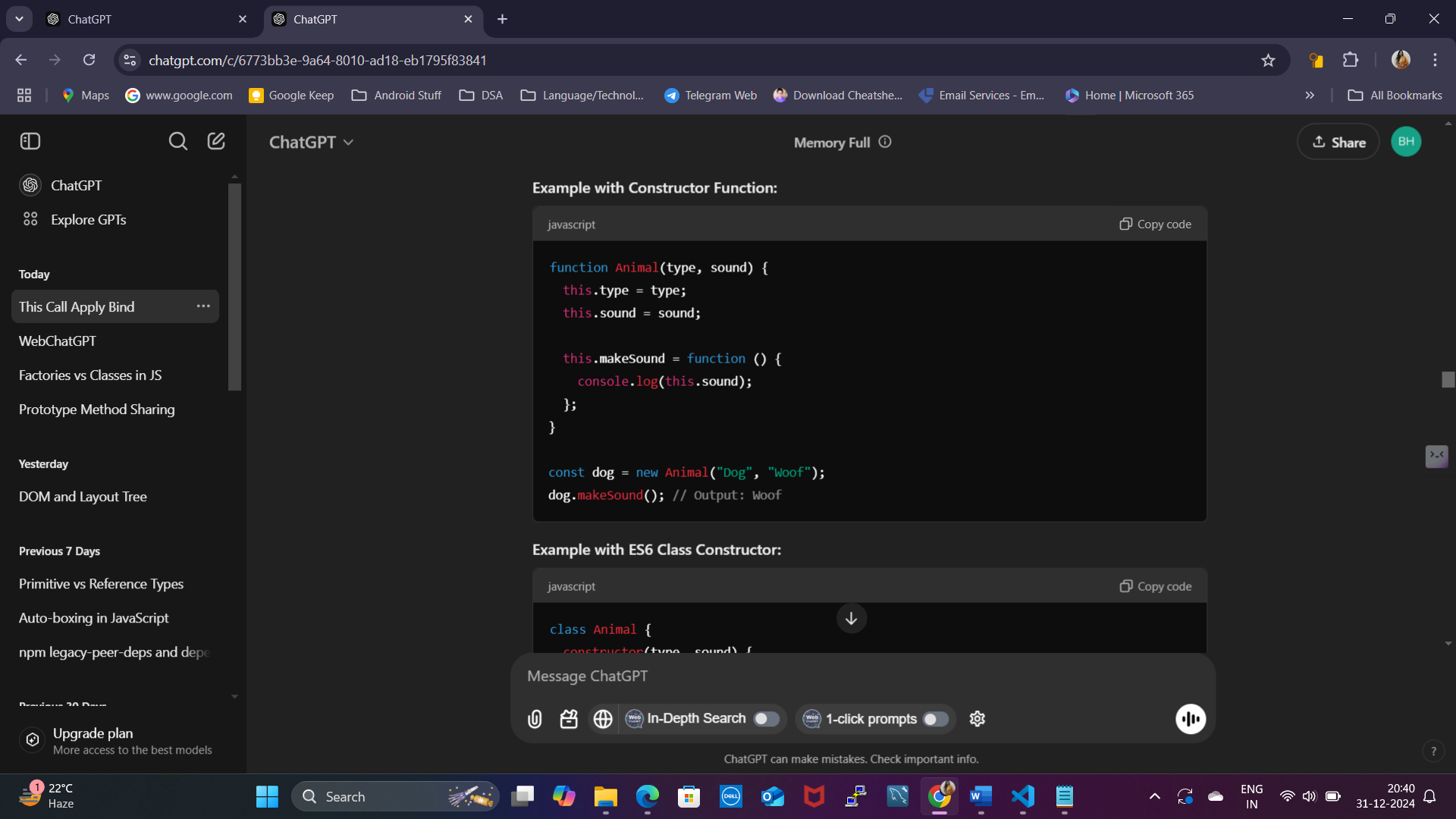
The new keyword is used to create an instance of an object from a constructor function or class. It does a few important things:

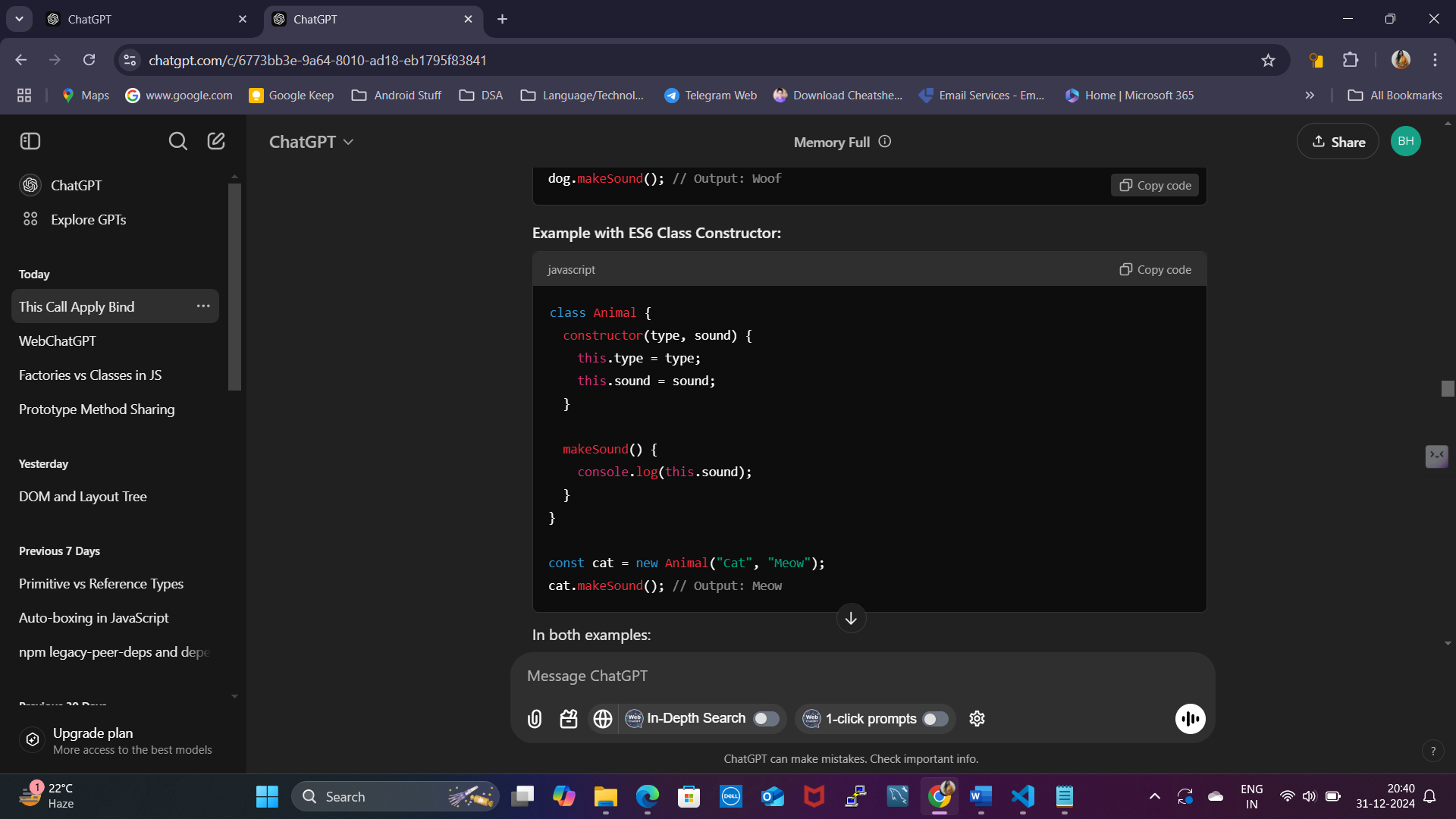
1. It creates an empty object.
2. It sets the prototype of the new object to the prototype of the constructor function.
3. It calls the constructor function with this set to the newly created object.
4. It returns the newly created object unless the constructor explicitly returns a different object.



**2. Constructor**

A constructor is a function (or a class) used to initialize objects created with the new keyword. It often sets properties and defines methods for the object.





**3. instanceof**

The instanceof operator checks whether an object is an instance of a constructor (or class). It looks up the prototype chain to see if the object inherits from the constructor's prototype.

4. Instance

An instance is the object created when you use the new keyword with a constructor. Each instance has its own copy of properties defined in the constructor.

**Prototype Sharing with Instances**

Instances share methods defined on the prototype, saving memory.

