**Concurrency Model:**

1. **Linker:**

* **Module Bundlers:** Tools like Webpack, Rollup, and Parcel bundle multiple JS files, resolving dependencies and ensuring all required modules are included in the final output.
* **ES Modules:** Allows importing and exporting functions, objects, or primitives between modules, linked together during the build process.

1. **Runtime:** The runtime environment is where JS code gets executed. The JS runtime includes:

* **Event Loop:** Core mechanism enabling non-blocking I/O operations by offloading to the system kernel.
* **Call Stack:** Tracks function calls, with functions pushed and popped as they are invoked and completed.
* **Task Queue:** Holds tasks **(callbacks**) for processing by the event loop, triggered by events like user interactions, network requests, or timers.
* **Microtask Queue:** Holds microtasks (e.g., promises) processed after the current script execution and before other tasks.

1. **Algorithm:** The algorithms related to concurrency in JS are primarily those that manage asynchronous operations and the event loop, such as:

* **Event Loop Algorithm:** This algorithm handles the process of picking up tasks from the task queue and executing them. It continuously checks the call stack and the task queue, ensuring that tasks are executed in the correct order.
* **Microtask Queue Processing:** After processing each task, the event loop examines the microtask queue and completes all microtasks before proceeding to the next task in the task queue.
* **Scheduling:** Various algorithms exist for scheduling tasks, such as setTimeout, setInterval, and requestAnimationFrame, which schedule tasks to run at future points in time.

1. **Pattern:** Concurrency patterns in JS include:

* **Callbacks:** Functions invoked after asynchronous operations are complete.
* **Promises:** Objects representing the completion or failure of an asynchronous operation.
* **Async/Await:** Syntactic sugar built on top of promises, allowing you to write asynchronous code that looks synchronous, improving readability and maintainability.
* **Observables:** Represent sequences of asynchronous events, used for handling events, data streams, etc.

1. **Dependency:** Dependency management in JS involves managing the relationships and dependencies between various modules and libraries in an application. This includes:

* **Module Systems**: Systems like CommonJS (used in Node.js) and ES Modules (standardized in ES6) manage module dependencies.
* **Package Managers:** Tools like npm and yarn manage library dependencies and versions, ensuring that all required packages are installed and correctly versioned.
* **Dependency Injection:** A pattern for providing objects with their dependencies to increase modularity and testability, often used in frameworks like Angular.

|  |  |  |
| --- | --- | --- |
| **Aspect** | **IIFE (Immediately Invoked Function Expression)** | **Self-Invoked Function** |
| **Definition** | A function that is both defined and executed immediately as an expression. | Any function that calls itself immediately upon definition. |
| **Syntax** | (function () {    // Code here runs immediately  })(); | // Function Expression  (function() {    // Code runs immediately  })();  // Function Declaration (not typically used this way)  function foo() {    foo();  }  foo(); |
| **Function Type** | Always a function expression. | Can be a function expression or function declaration. |
| **Purpose** | Create a new scope to encapsulate variables and prevent global scope pollution. | Can be used for encapsulation, but also includes recursion and immediate invocation. |
| **Usage** | Primarily for encapsulation and modularization. | Includes recursion and any immediate invocation scenarios. |
| **Scope** | Creates a new local scope. | May or may not create a new local scope depending on context. |
| **Common Use Cases** | Encapsulation, module pattern, initialization code. | Recursive functions, immediate execution of setup code. |
| **Example (Expression)** | (function () {    var message = "Hello, IIFE!";    console.log(message);  })(); | (function () {    var message = "Hello, Self-Invoked Function!";    console.log(message);  })(); |
| **Example (Declaration)** | N/A | function recursiveSelfInvoke(count) {    if (count > 0) {      console.log("Count:", count);      recursiveSelfInvoke(count - 1);    }  }  recursiveSelfInvoke(3); |
| **Emphasis** | Immediate invocation and encapsulation. | Immediate invocation in a broader sense. |

**Summary**

* **IIFE**: A specific kind of self-invoked function that is always a function expression, used to create new scopes and prevent global variable pollution.
* **Self-Invoked Function**: A broader term that refers to any function that calls itself immediately upon definition, which can be a function expression or a function declaration.

**Arrow Function Doesn't Bind This Keyword and Arguments Object**

1. **"this" Binding**:

* **Lexical Scope for "this"**: In arrow functions, **"this"** is inherited from the surrounding context where the arrow function is defined.
* **No New "this" Binding**: Unlike regular functions, arrow functions do not create a new **"this"** context. They use the **"this"** value from the enclosing non-arrow function or the global context if no such function exists.
* **Example: "this" Binding**:

// Traditional function

function traditionalFunction() {

  console.log(this); // Depends on how the function is called

}

// Arrow function

const arrowFunction = () => {

  console.log(this); // Lexical this

};

const obj = {

  traditionalMethod: function () {

    console.log(this); // Points to obj

    setTimeout(function () {

      console.log(this); // Points to global object or undefined in strict mode

    }, 100);

  },

  arrowMethod: function () {

    console.log(this); // Points to obj

    setTimeout(() => {

      console.log(this); // Inherits obj's this context

    }, 100);

  },

};

// Usage

obj.traditionalMethod();

obj.arrowMethod();

1. **"arguments" Object**:

* No Own **"arguments"** Object: Arrow functions do not have their own **"arguments"** object. If **"arguments"** is referenced within an arrow function, it refers to the **"arguments"** object of the nearest non-arrow function.
* Use Rest Parameters: To access the arguments passed to an arrow function, use the rest parameter syntax (**"...args"**).
* **Example: "arguments" Object**:

const arrowFunction = () => {

  // console.log(arguments); // Error: arguments is not defined

};

const arrowFunctionWithRest = (...args) => {

  console.log(args); // Use rest parameters to access arguments

};

arrowFunctionWithRest(1, 2, 3); // logs [1, 2, 3]

|  |  |  |
| --- | --- | --- |
| **Feature** | **Traditional Function** | **Arrow Functions** |
| **"this" Binding** | Determined by how the function is called. | Lexically inherited from the surrounding scope. |
| **"arguments" Object** | Has its own **"arguments"** object containing passed arguments. | Does not have its own **"arguments"** object; use rest parameters **("...")**. |

**let x = 50; // undefined => Why?**

const obj = {

  x: 10,

  getX: () => {

console.log(this.x);

  },

};

obj.getX();

Because the **"getX"** method is an arrow function, it uses the **"this"** binding from its surrounding scope at the time it was created. During creation, **"this"** refers to the global object (window in a browser), not the **"obj"** object. Since there is no global variable **"x"**, the output is **"undefined"**.