**“This” keyword**

1. **Explain the different values `this` can take on in the global scope depending on the JavaScript environment (browser vs. Node.js)**
2. **Browser Environment:**

* **Global Scope (outside any function):** In the global scope, this refers to the window object in a browser. The window object represents the global context of the web page.

Ex:- console.log(this); // browser: globalThis -> window

* **Inside a Function (non-strict mode):** When this is used inside a regular function (not an arrow function) in non-strict mode, it still refers to the window object.

Ex:- function f1() {

console.log(this); // window

}

f1();

* **Inside a Function (strict mode):** In strict mode, this inside a function is undefined if the function is called in the global context.

Ex:- 'use strict';

function f1() {

console.log(this); // undefined

}

f1();

**2. Node.js Environment:**

* **Global Scope (outside any function):** In Node.js, this in the global scope does **not** refer to the global object (which is the Node.js equivalent of the window object in browsers). Instead, this is an empty object {}.

Ex:- console.log(this); // cjs: {} esm: undefined

* **Inside a Function (non-strict mode):** When this is used inside a regular function in non-strict mode, it refers to the global object.

Ex:- function f1() {

console.log(this); // global object

}

f1();

* **Inside a Function (strict mode):** Similar to the browser, in strict mode, this inside a function is undefined if the function is called in the global context.

Ex:- 'use strict';

function f1() {

console.log(this); // undefined

}

f1();

1. **Describe the behavior of `this` inside regular functions under both strict and non-strict modes.**

**1. Non-Strict Mode:**

* **Global Context:**
  + When a regular function is invoked in the global context (i.e., not as a method of an object), this refers to the global object.
  + In the browser environment, this global object is window.
  + In the Node.js environment, this global object is global.

Ex:- function showThis() {

console.log(this);

}

showThis(); // In the browser: window, In Node.js: global

* **Object Context:**
* When a function is called as a method of an object, this refers to the object through which the function was called.

Ex:- const obj = {

name: 'Example',

showThis: function() {

console.log(this);

}

};

obj.showThis(); // Logs the obj object

* **Function Called with call or apply:**
* If a function is called using call() or apply(), this is explicitly set to the first argument passed to call or apply.

Ex:- function showThis() {

console.log(this);

}

const obj = { name: 'Example' };

showThis.call(obj); // Logs the obj object

showThis.apply(obj); // Logs the obj object

* **Constructor Functions:**
* When a function is used as a constructor (called with the new keyword), this refers to the newly created instance object.

Ex:- function Person(name) {

this.name = name;

}

const person = new Person('John');

console.log(person.name); // 'John'

**2. Strict Mode:**

* **Global Context:**
* In strict mode, when a regular function is invoked in the global context, this is undefined. This is because strict mode prevents this from defaulting to the global object.

Ex:- 'use strict';

function showThis() {

console.log(this);

}

showThis(); // undefined

* **Object Context:**
* In strict mode, when a function is called as a method of an object, this still refers to the object through which the function was called, just like in non-strict mode.

Ex:- 'use strict';

const obj = {

name: 'Example',

showThis: function() {

console.log(this);

}

};

obj.showThis(); // Logs the obj object

* **Function Called with call or apply:**
* In strict mode, if a function is called using call() or apply(), this will be exactly what you pass as the first argument, even if it’s null or undefined. If null or undefined is passed, this remains null or undefined instead of defaulting to the global object.

Ex:- 'use strict';

function showThis() {

console.log(this);

}

showThis.call(null); // null

showThis.apply(undefined); // undefined

* **Constructor Functions:**
* Strict mode doesn’t change how this behaves in constructor functions. It still refers to the newly created instance.

Ex:- 'use strict';

function Person(name) {

this.name = name;

}

const person = new Person('John');

console.log(person.name); // 'John'

1. **What is "this substitution" in the context of non-strict mode JavaScript functions, and how does it work?**

In the context of non-strict mode JavaScript functions, "this substitution" refers to the automatic process by which JavaScript assigns a value to this when a function is invoked in a way that doesn't explicitly define what this should be. This behavior occurs mainly in the global context and when functions are invoked without an explicit object.

* **No Substitution in Strict Mode:** In strict mode, this substitution does not occur. Instead, this remains undefined if it's not explicitly set.
* **Object Method Context:** If a function is called as a method of an object, this refers to that object, and no substitution occurs, even in non-strict mode.
* **How "This Substitution" Works:**

**Global Context (Non-Strict Mode):**

* + In non-strict mode, if a regular function is called in the global context (i.e., not as a method of an object), and without an explicit receiver (e.g., someObject.someMethod()), JavaScript automatically sets this to the global object.
  + **In the Browser:** The global object is window.
  + **In Node.js:** The global object is global.

Ex:- function showThis() {

console.log(this);

}

showThis(); // In the browser: window, In Node.js: global

**Implicit Undefined "This" (Non-Strict Mode):**

* When a function is invoked without an object context, JavaScript checks what this is. If this is undefined (which it would be in strict mode or in other specific situations), non-strict mode automatically substitutes this with the global object.

Ex:- function showThis() {

console.log(this);

}

var showThisGlobal = showThis;

showThisGlobal(); // In non-strict mode, `this` becomes the global object.

* **Examples of "This Substitution":**
* **Function Called in the Global Context:**

Ex:- function greet() {

console.log(this);

}

greet(); // Logs `window` in browsers or `global` in Node.js because of "this substitution."

* **Assigning a Function to a Variable:**

Ex:- function display() {

console.log(this);

}

var show = display;

show(); // Logs the global object due to "this substitution" in non-strict mode.

* **Calling a Function as a Method of null or undefined:**
* If you explicitly try to invoke a function with null or undefined as this, in non-strict mode, JavaScript will substitute this with the global object.

Ex:- function logThis() {

console.log(this);

}

logThis.call(null); // Logs `window` in browsers or `global` in Node.js.

logThis.call(undefined); // Logs `window` in browsers or `global` in Node.js.

1. **How does `this` behave within arrow functions compared to regular functions? Explain the concept of lexical scope in relation to this behavior.**

**1. Arrow Functions:**

* Lexical this:
  + Arrow functions do not have their own this context. Instead, this in an arrow function is lexically bound, meaning it captures the this value from the surrounding (enclosing) scope where the arrow function is defined.
  + The value of this in an arrow function is permanently set to the value of this in its surrounding context at the time of the arrow function's creation, and it cannot be changed.
  + This behavior makes arrow functions particularly useful for maintaining the correct this value inside callbacks or in situations where the context changes.

Ex:- const obj = {

name: 'Object',

regularFunction: function() {

console.log(this.name); // 'Object'

},

arrowFunction: () => {

console.log(this.name); // The value of `this` is inherited from the surrounding scope.

}

};

obj.regularFunction(); // 'Object'

obj.arrowFunction(); // Depending on the context, this could be undefined or something else.

Example of Lexical this:

function Outer() {

this.name = 'Outer';

const arrowFunc = () => {

console.log(this.name); // 'Outer'

};

arrowFunc();

}

const outer = new Outer();

**2. Regular Functions:**

* Dynamic this:
  + Regular functions have their own this context, which is determined dynamically based on how the function is called.
  + In non-strict mode, if a regular function is called in the global context or as a standalone function, this refers to the global object (window in browsers, global in Node.js).
  + In strict mode, this is undefined when the function is called in the global context or without an object context.
  + When called as a method of an object, this refers to that object.

Ex:- const obj = {

name: 'Object',

regularFunction: function() {

console.log(this.name); // 'Object'

}

};

obj.regularFunction(); // 'Object'

Example of Dynamic this:

const obj = {

name: 'Object',

regularFunction: function() {

console.log(this.name); // 'Object'

}

};

const anotherObj = { name: 'Another Object' };

obj.regularFunction.call(anotherObj); // 'Another Object' (because of dynamic `this` binding)

**3. Lexical Scope and this:**

* **Lexical Scope:**
  + Lexical scope refers to the scope in which variables are accessible based on where they are written in the code. It is determined by the physical structure of the code (where functions and variables are defined), not how or when functions are called.
  + Arrow functions inherit this from their lexical scope, meaning they do not bind their own this but rather use this from the scope in which they were defined.
* **Impact on this:**
  + In regular functions, this is determined by the execution context, which can vary depending on how the function is called.
  + In arrow functions, this is determined by the lexical scope, making it consistent and predictable because it is set at the time of function definition and cannot be altered.

1. **In the context of object methods, what does this typically refer to? Provide an illustrative example.**

In the context of object methods, this typically refers to the object through which the method was called. This allows the method to access and manipulate the properties and other methods of the object.

**Illustrative Example:**

Consider an object person that has properties like firstName and lastName, and a method getFullName that uses this to access those properties.

Ex:- const person = {

firstName: 'John',

lastName: 'Doe',

getFullName: function() {

return this.firstName + ' ' + this.lastName;

}

};

// Calling the method

console.log(person.getFullName()); // Outputs: 'John Doe'

* **Explanation:**
* **Object Definition:** The person object has two properties (firstName and lastName) and one method (getFullName).
* **Method Invocation:** When person.getFullName() is called, the method getFullName is executed with this referring to the person object.
* **Accessing Properties:** Inside the getFullName method, this.firstName accesses the firstName property of the person object, and this.lastName accesses the lastName property.
* **Output:** The method returns the full name by combining firstName and lastName, resulting in 'John Doe'

1. **Explain how call, apply, and bind methods can manipulate the value of `this` within a function.**

**1. call Method:**

* **Purpose:** The call method allows you to invoke a function immediately, specifying what this should be within that function. Additionally, you can pass arguments to the function individually.
* **Syntax:** function.call(thisArg, arg1, arg2, ...)
* **Parameters:**
  + thisArg: The value of this that you want to set for the function.
  + arg1, arg2, ...: Arguments passed to the function.

Ex:- function greet(greeting, punctuation) {

console.log(greeting + ', ' + this.name + punctuation);

}

const person = { name: 'Alice' };

greet.call(person, 'Hello', '!'); // Outputs: 'Hello, Alice!'

**2. apply Method:**

* **Purpose:** The apply method is similar to call, but it accepts an array (or array-like object) of arguments instead of passing them individually. This is useful when the number of arguments is unknown or dynamic.
* **Syntax:** function.apply(thisArg, [argsArray])
* **Parameters:**
  + - thisArg: The value of this that you want to set for the function.
    - [argsArray]: An array or array-like object containing the arguments to pass to the function.

Ex:- function greet(greeting, punctuation) {

console.log(greeting + ', ' + this.name + punctuation);

}

const person = { name: 'Bob' };

const args = ['Hi', '.'];

greet.apply(person, args); // Outputs: 'Hi, Bob.'

**3. bind Method:**

* **Purpose:** The bind method does not immediately invoke the function. Instead, it creates a new function with this permanently set to the value you specify. You can also pre-define some arguments, which will be prepended to any arguments provided when the bound function is called.
* **Syntax:** function.bind(thisArg, arg1, arg2, ...)
* **Parameters:**
  + thisArg: The value of this that you want to set for the function.
  + arg1, arg2, ...: (Optional) Arguments to be pre-filled in the function.

Ex:- function greet(greeting, punctuation) {

console.log(greeting + ', ' + this.name + punctuation);

}

const person = { name: 'Charlie' };

const boundGreet = greet.bind(person, 'Hey');

boundGreet('!!'); // Outputs: 'Hey, Charlie!!'

* **Key Differences:**
* **call and apply:**
  + Both methods immediately invoke the function with a specified this value.
  + call passes arguments individually, while apply expects an array of arguments.
* **bind:**
  + bind creates a new function with a permanently set this value and optionally pre-filled arguments.
  + The new function can be called later with additional arguments.