This keyword

`this` is a keyword that refers to the current object or context in which it is used. It is used to access the properties and methods of the current object.

1.Global Context: When **this** is used in the global scope (outside of any function), it refers to the global object. In a web browser environment, the global object is **window**.

```
console.log(this === window); // true
```

1. **Function Context**: When **this** is used within a function that is not a method of an object, its value depends on how the function is called. If the function is called as a standalone function, **this** will typically refer to the global object (or **undefined** in strict mode).

```
 function sayHello() {
  console.log(this);
}

sayHello(); // In a browser, this would log the window object
```

2.**Method Context**: When **this** is used within a method of an object, it refers to the object that the method is called on.

//here this refers to the person means object name

```
const person = {
 name: 'John',
 greet: function() {
```

```
        console.log('Hello, my name is ' + this.name);
    }
};

person.greet(); // Logs "Hello, my name is John"
```

This in arrow function

```
    let obj={
        name:"John",
        age:30,
        sayHello:()=>{
            console.log(this.age) //undefined
        }
    }

    obj.sayHello()
```

```
    let obj={
        name:"John",
        age:30,
        sayHello:()=>{
            console.log(this)// window
        }
    }

    obj.sayHello()
```

Understanding this in Arrow Functions

Arrow functions (() => { }) in JavaScript behave differently with respect to this compared to regular functions. Here are the key points to note:

1. **Lexical Scope**: Arrow functions do not have their own this context. Instead, they inherit this from the surrounding (lexical) scope where they are defined.
2. **Global Object**: In most cases where obj is defined at the top level (not inside another function or block), this refers to the global object (window in browsers).

This refers to the particular event in event handlers

```
 document.getElementById('myButton').onclick = function() {
 console.log(this); // Refers to the button element with the ID
'myButton'
   };
```

Changing this with call, apply, and bind
call and **apply** immediately invoke the function with a specified this value and arguments.

```
function showThis() {
  console.log(this);
}
const obj = { name: "John" };

showThis()// window
showThis.call(obj); // obj
showThis.apply(obj; // obj
```

bind
bind returns a new function with a specified this value, without immediately invoking the function.

```
const boundFunction = showThis.bind(obj);
boundFunction(); // obj
```

Call With Parameters

```
function hello(city, profession) {
 console.log(
   "hello my name is " +
```

```
      this.name +
      " iam from " +
      city +
      " my profesion is " +
      profession
  );
}

obj = {
  name: "john",
};

obj2 = {
  name: "peter",
};

hello.call(obj, " vizag", " senior trainer");
hello.call(obj2, "hyd", " developer ");
```

Apply

```
function hello(city, profession) {
  console.log(
    "hello my name is " +
      this.name +
      " iam from " +
      city +
      " my profesion is " +
      profession
  );
}
obj = {
```

```
  name: "john",
};

obj2 = {
  name: "peter",
};

hello.apply(obj, [" vizag", " senior trainer"]);
hello.apply(obj2, ["hyd", " developer "]);
```

bind

```
function hello(city, profession) {
  console.log(
    "hello my name is " +
      this.name +
      " iam from " +
      city +
      " my profesion is " +
      profession
  );
}

obj = {
  name: "john",
};

var bind1 = hello.bind(obj);
bind1(" vizag", " senior trainer");
```

Key Differences and Use Cases

call vs apply: The primary difference between call and apply is how arguments are passed:

- call passes arguments individually.
- apply expects an array of arguments.

Use call when you know the exact arguments to pass, and use apply when you have arguments in an array or array-like object.

bind: Unlike call and apply, bind does not immediately invoke the function. Instead, it creates a new function with the bound this value and optionally prepended arguments. This is useful when you want to create a function with a fixed this value that you can later execute.

Constructor function

It is a old way to create objects

Single same objects

```
function Person(){
    this.name = "john";
    this.age = 30;
}



let person1=new Person();

console.log(person1);
let person2=new Person();
console.log(person2);
```

Multi objects

```
// constructor function - it is a old way to create objects in js
  function Person(name, age){
      this.name = name;
      this.age = age;
  }

  let person1=new Person("Hemanth", "26");
  console.log(person1);
  let person2=new Person("teja", "25");
  console.log(person2);
```

Prototype

```
  function Person(){
      this.name = "john";
      this.age = 30;
  }

  Person.prototype.course="trainers"

  let person1=new Person();
  console.log(person1.course);
  let person2=new Person();
  console.log(person2.course)
```

function Person(name, age) {

```
  this.name = name;
  this.age = age;
}

function Student(name, age, grade) {
  // Using call to inherit the properties from Person
  Person.call(this, name, age);
  this.grade = grade;
}

const student1 = new Student('John', 20, 'A');
console.log(student1); // Output: { name: 'John', age: 20, grade: 'A'
}
```