This keyword

The value of `this` is determined by how a function is called (runtime binding). So, there are two types of binding when it comes to object binding in JS, one is implicit and other is explicit.

Implicit Binding

Implicit Binding is applied when you invoke a function in an Object using the dot notation. **this** in such scenarios will point to the object using which the function was invoked. Or simply the object on the left side of the dot.

Explicit Binding

In Explicit Binding, you can force a function to use a certain object as its this. Explicit Binding can be applied using call(), apply(), and bind().

Question 1 - Explain 'this' keyword?

In the English language, we use the pronoun 'this' to reference something:

Like suppose we have a bucket of fruits, when we say "this" inside of it so that will mean the bucket itself.

Fruits are kept in this bucket

Similarly, in the JavaScript language, the 'this' keyword is used to reference something — an object!

It can't be set by assignment during execution, and it may be different each time the function is called.

Global

So for example if we console.log `this` here, we get the window object.

```
let a = 5

console.log(this.a); //undefined
```

This inside a function

- Normally it targets the window object.

In it, `this` points to the owner of the **function call**, I repeat, THE FUNCTION CALL, and NOT the function itself. The same function can have different owners in different scenarios.

```
function myFunction() {
    console.log(this)
}
myFunction(); // window object
```

What about Arrow Functions?

It takes it's `this` from the outer “normal” function, it’s also pointing to window object.

```
const myFun=()=> {
    console.log(this)
}
myFun(); // window object
```

So lets see the behaviour of `this` inside of an Object

```
let user = {
  name: "Hayat",
  age: 24,
    getDetails() {
        console.log(this.name); //output: Hayat
    }
};
```

What happens when we have functions inside a nested object key?

```
let user = {
    name: "Piyush",
    age: 24,
      childObj:{
          newName:"Lazy Coder",
          getDetails() {
              console.log(this.newName, "and" ,this.name);
          }
      }
  };

  user.childObj.getDetails() //output: Lazy Coder and undefined
```

What if the same functions are arrow functions inside the object?

```
let user = {
    name: "Piyush",
    age: 24,
      getDetails: () => {
          console.log(this.name);
      }
  };

  user.getDetails()//no output.t is empty since it points to window object.
```

```
let user = {
    name: "Hayat",
```

```
    age: 24,
      getDetails() {
          const nestedArrow = () => console.log(this.name); //Output:Hayat
          nestedArrow();
      }
  };
  user.getDetails()
//user.getDetails() gives "Hayat" as the output since it points to the parent's context i.e. the user object.
```

Class / Constructors

```
class user {
    constructor(n){
        this.name = n
    }
    getName(){
        console.log(this.name);
    }
}

const User = new user("Piyush") // => This will generate a user object from the above class
User.getName();

//in class this will point to the constructor
```

Question-01: Give the output of the following snippet.

```
const user = {
    firstName: 'Hayat',
    getName() {
      const firstName = 'Hossain';
      return this.firstName;
    }
  };
  console.log(user.getName()); // output: Hayat//this will always point to parent object
```

Question-02: What is the result of accessing its `ref`? Why?

```
function makeUser() {
  return {
    name: "John",
    ref: this
  };
}

let user = makeUser();

console.log( user.ref.name );//no output // no parent object so ref will point to window object
```

to fix it we have to make ref function

```
function makeUser() {
  return {
    name: "Hayat Hossain",
    ref() {
      return this;//no this will point to parent object
    }
  };
}

let user = makeUser();

console.log( user.ref().name ); // output: Hayat Hossain
```

Question-03: What is the output

```
const user = {
    name: "Hayat Hossain",
    logMessage() {
        console.log(this.name)
    }
}
setTimeout(user.logMessage, 1000)
//output: empty
//it is because setTimeout using user.logMessage as a
//callback rather than a method so this complete function
// right here will be copied inside this settimeout and this
//will no longer have access to the user object.since it
//executing independently in this case this will have the
//access to the window object
```

To fix it we have to wrap it with a function. so it will point to the user object.

```
const user = {
    name: "Hayat Hossain",
    logMessage() {
        console.log(this.name)
    }
}
setTimeout(function () { user.logMessage() }, 1000)
//output: Hayat Hossain
```

Question-04: const result = calc.add(10).multiply(5).subtract(30).add(10) console.log(result.total) ---- What is logged?

```
var calc = {
    total: 0,
    add(a) {
        this.total += a;
        return this;
    },
    subtract(a) {
        this.total -= a;
```

```
        return this;
    },
    multiply(a) {
        this.total *= a;
        return this;
    },
};
```