`this` in Functions & Classes

Understanding `this` in arrow function (=>) and regular function (function)

`this` and Its Interpretation

Imagine You have a toy box.

When you say:

> “Show me **my** toy!”

someone needs to know who “me” is.

That’s what `this` means in JavaScript — it’s like saying “me”.

Imagine you have a toy box again.
When you say to arrow function (=>):

> “Show me **my** toy!”

But arrow functions don’t ask anything about "me" — arrow functions **borrow “me”** from where they (arrow functions) were born.

That’s what makes them special.

Arrow Function Example

```
const person = {
  name: "Alice",
  normal: function() { // will be created when it rusn
    console.log("Normal:", this.name);
  },
  arrow: () => { // will be created when it runs
    console.log("Arrow:", this.name);
  }
};

person.normal();
person.arrow();
```

What happens when you write this code and when this file runs:

1. JavaScript creates the object person (dynamically at runtime).
2. It fills it with:

- a property name: "Alice"
- a property normal containing a regular function
- a property arrow containing an arrow function

3. The person object is dynamically created in memory when the code runs.

- But that’s not the tricky part — the tricky part is how this is set when you call the functions.

For person.normal()

When you call: `person.normal();`

JavaScript looks at the left side of the dot (person) and says:

> “Okay, person is the one calling this function — so this = person.”

Therefore, inside normal(), `this.name = "Alice"`.

For person.arrow()

When you call: `person.arrow();`

Arrow functions don’t care who calls them!

- They keep their `this` from where they were created (the lexical scope).
- In this case, the arrow function was created when the script loaded, and the surrounding scope was the global (or module) scope — not the person object.

So:

- The arrow function’s this = global / undefined
- this.name → undefined

Analogy: Birth vs Adoption

- Function Type “Who am I?” decided by: Who calls me (adopted parent) “I live with whoever called me.”
- Arrow function becided by: Where I was born (lexical parent) “I always remember where I was born.”

So even if person.arrow is stored inside person,
it remembers its birthplace — the outer/global scope.

What Happens

```
person.normal(); // this.name?
person.arrow(); // this.name?
```

When you run this:

```
Normal: Alice
Arrow: undefined
```

Why?

- The normal function finds its parents (adoption), and it is "person" object, so `person.name` is returned.
- The arrow function knows where it is born (birth), and it is the "global = {}", there is no "name" in global, so `{}.name == undefined` is returned.

Function Example

Without "use strict";

```
function showThis() {
  console.log(this);
}
const showThis2 = () => {
  console.log(this);
};
```

- showThis (adaption) thinks `this` is the Node.js (or Webbrowser), so the whole `Node.js object` is printed.
- showThis2 (birth) thinks `this` is global, so it prints out `{}` (empty global variables).

With "use strict";

- showThis (adaption) does not allow to assume `this` is the Node.js (or Webbrowser), so `undefined` is printed.
- showThis2 (birth) thinks `this` is global, so it prints out `{}` (empty global variables).

Inner function Example (Lexical Scoping)

Normal Function (function)

```
const person = {
  name: "Alice",
  greet: function() {
    console.log("Outer:", this.name); // "Alice"
    // ❌ undefined (or global object)
    function inner() {console.log("Inner:", this.name);
    }
    inner();
  },
};
person.greet();
```

- Regular functions have their own this.
- When you call inner() directly as in this example,

JavaScript says:

> “No one called me — I don’t know who I belong to!”

- In non-strict: it defaults to the global object {}, so undefined.
- In strict mode: this → undefined, so error!

Arrow Function (=>)

```
const person2 = {
  name: "Alice",
  greet: function() {
    console.log("Outer:", this.name); // "Alice"

    const inner = () => {
      console.log("Inner:", this.name); // ✅ "Alice"
    };

    inner();
  },
};
```

Arrow functions don’t have their own this.

- Instead, they borrow this from the place they were created: Inside person2.greet(), which is a regular function where this already points to person2.
- So the arrow function’s this = same as its parent (person2).

Database Example

In database, we sometimes need to use `this` for accessing information.

- In this case, we must use regular function, not lambda expression.

```
// ❌ Arrow function — won't work with lastID
db.run(sql, params, (err) => {
  console.log(this.lastID); // undefined
});
```

```
// ✅ Regular function — correct "this" binding
db.run(sql, params, function(err) {
  console.log(this.lastID); // Works: e.g., 5
});
```

In short

- When we don't use `this`, prefer to use lambda expression for simplicity.
- When we use `this`, prefer to use regular function, as lambda expression does not know `this` from the lexical scoping.