```
const obj = {
  name:'Rohit',
  arrowChangeName: (name) => {
    this.name=name
    console.log(this)
  },
  normalChangeName: function (name) {
    this.name=name
    console.log(this)
  }
}

obj.arrowChangeName('mohit')
console.log(obj.name)
obj.normalChangeName('mohit')
console.log(obj.name)
```

Output:

```
{ name: 'mohit' }
Rohit
{
  name: 'mohit',
  arrowChangeName: [Function: arrowChangeName],
  normalChangeName: [Function: normalChangeName]
}
mohit
```

Here is a step-by-step breakdown of your example, including the output, and an explanation of the core difference in `this` binding between the two function types. The key distinction is that a normal function has a `this` value that is dynamically determined by how it's called, while an arrow function inherits its `this` value lexically from its parent scope.

The code

javascript

```javascript
const obj = {
  name: 'Rohit',
  arrowChangeName: (name) => {
    this.name = name;
    console.log(this);
  },
```

```
  normalChangeName: function (name) {
    this.name = name;
    console.log(this);
  },
};

obj.arrowChangeName('mohit');
console.log(obj.name);
obj.normalChangeName('mohit');
console.log(obj.name);
```
Use code with caution.

Execution breakdown and output

Step 1: `obj.arrowChangeName('mohit')`

- The `arrowChangeName` method is an arrow function.
- Arrow functions do not bind their own `this`. Instead, they inherit the `this` value from their enclosing lexical (parent) scope.
- In this case, the arrow function is defined in the top-level scope (the global scope), where `this` refers to the global object.
- Executing `this.name = name` will create a new `name` property on the global object (e.g., the `window` object in browsers or `global` in Node.js), and assign it the value `'mohit'`. It will **not** modify `obj.name`.
- `console.log(this)` will log the global object, which now has a `name` property.

Output (for this step):

```
<ref *1> Object [global] {
  ...
  name: 'mohit'
}
```

Step 2: `console.log(obj.name)`

- This line prints the `name` property of the `obj` object.
- The arrow function in the previous step did not change `obj.name`, so it still holds its original value.

Output (for this step):

```
Rohit
```

Step 3: `obj.normalChangeName('mohit')`

- The `normalChangeName` method is a standard function.
- When a normal function is invoked as a method of an object (using the dot notation `obj.normalChangeName()`), its `this` value is dynamically bound to the object that owns the method—in this case, the `obj` object.
- Executing `this.name = name` will change the `name` property of the `obj` object to `'mohit'`.
- `console.log(this)` will log the `obj` object itself.

Output (for this step):

```
{ name: 'mohit', arrowChangeName: [Function], normalChangeName: [Function:
normalChangeName] }
```

Step 4: `console.log(obj.name)`

- This line prints the `name` property of the `obj` object again.
- This time, the value has been changed by the normal function call.

Output (for this step):

```
mohit
```

Full combined output

When you run the code, you would see the following:

```
<ref *1> Object [global] {
  ...
  name: 'mohit'
}
Rohit
{ name: 'mohit', arrowChangeName: [Function], normalChangeName: [Function:
normalChangeName] }
mohit
```

what is a lexical parent and why this inside arrow function in above example refers to global scope when arrow function is defined in the obj object's local scope

This is a very common point of confusion in JavaScript. The arrow function inside your `obj` literal refers to the global scope because a plain object literal (`{ ... }`) does **not** create a new scope for `this`. The arrow function inherits the `this` from its immediate parent, which is the global scope.

What is a lexical parent?

A lexical parent refers to the static, or physical, relationship between nested code blocks. It's about where code is **written**, not where it is **called**. When the JavaScript engine looks for a variable (or `this` in an arrow function), it first checks the current lexical scope. If it's not found, it moves up the "scope chain" to the parent lexical scope, and so on, until it reaches the global scope.

In your example:

javascript

```
// Global scope
const obj = {
  name: 'Rohit',
  arrowChangeName: (name) => {
    // Parent lexical scope is the global scope
    this.name = name;
    console.log(this);
  }
};
```

Use code with caution.

The **lexical parent** of the `arrowChangeName` method is the top-level global scope, not the `obj` object itself.

Why the arrow function's `this` is the global object

1. **Object literals don't have scope:** A simple object literal (`{}`) does not create its own execution context or `this` binding. Its properties exist within the scope where the object was defined.
2. **Arrow functions are lexically scoped:** An arrow function captures the value of `this` from its nearest enclosing (parent) lexical scope at the time it is defined.
3. **Inheritance from global:** In your code, the arrow function is defined in the top-level global scope. Therefore, it inherits the global object (`window` in a browser or `global` in Node.js) as its `this` value.

4. **No implicit binding:** The way the arrow function is called (`obj.arrowChangeName()`) has no effect on its `this` value. This is the crucial difference from a normal function, which has its `this` dynamically bound to the calling object.

What about a regular function?

In contrast, the `normalChangeName` method is a standard function. When it is invoked with the `obj.normalChangeName()` syntax, JavaScript dynamically binds its `this` value to the object on which the method is called. This is why `this` correctly refers to `obj` in the normal function.

How to use an arrow function with the correct `this`

If you want an arrow function to have `this` refer to the object it belongs to, you must define it inside a regular function or a class that creates its own `this` binding.

Example with a regular function parent:

javascript

```javascript
const objWithCorrectArrow = {
  name: 'Rohit',
  // This normal function creates its own `this` binding
  normalMethod: function() {
    // The arrow function inside inherits `this` from the `normalMethod`,
    // which is dynamically bound to `objWithCorrectArrow` when called.
    const arrowChangeName = (name) => {
      this.name = name;
      console.log(this);
    };
    arrowChangeName('mohit');
  }
};

objWithCorrectArrow.normalMethod();
// Output: { name: 'mohit', normalMethod: [Function: normalMethod] }
```