The `this` keyword inside an arrow function behaves differently compared to traditional function expressions in JavaScript.

Key characteristics of `this` in arrow functions:

- Lexical `this` binding:

  Arrow functions do not have their own `this` context. Instead, they inherit the `this` value from the enclosing lexical scope where they are defined. This means `this` inside an arrow function refers to the `this` of the surrounding code, not the object on which the function is called or the global object (in strict mode).

- **No re-binding:**
  The `this` value of an arrow function cannot be re-bound using methods like `call()`, `apply()`, or `bind()`. It remains fixed to the `this` of its defining scope.

Implications:

- **Preserving context:** This lexical `this` binding is particularly useful in scenarios like callbacks or nested functions within object methods, where you want to maintain the `this` context of the parent scope.

JavaScript

```javascript
const obj = {
  name: 'Alice',
  greet: function() {
    // 'this' here refers to 'obj'
    const arrowGreet = () => {
      console.log(`Hello, ${this.name}!`); // 'this' still refers to 'obj'
    };
    arrowGreet();
  }
};
obj.greet(); // Output: Hello, Alice!
```

Event handlers: When using arrow functions as event handlers, `this` will refer to the context where the event listener was defined, not the element that triggered the event. If you need to access the event target, you would typically use the `event.target` property within the event handler.