Function Relationship with "this"

In this lesson, you will learn about the relationship of functions with the "this" keyword.

WE'LL COVER THE FOLLOWING

Function and `this`

TypeScript has the capability of letting the author of a function specify the type of `this`. Without it, even with the arrow function, this may be of type `any`. The lack of a type on this often occurs if you are using an object literal has a function that returns a function that uses `this`. The function in the object literal is called a function expression and returns `any` type when the value is accessed with `this`.

However, we know that the type means the object literal, hence we can specify to TypeScript the type of this one. The syntax is to use `this` followed by a colon symbol followed by the expected type as the first parameter of the function signature.

In the following example, **line 12** returns `any` when you may expect a `string` since the type at **line 2** is an array of `string`.

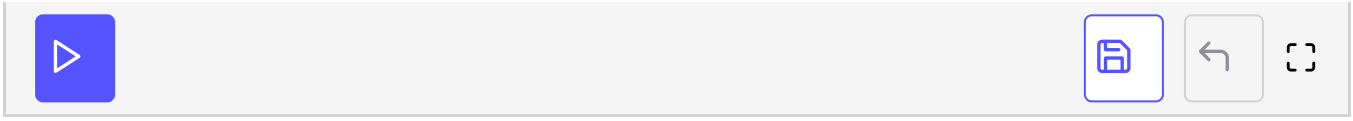
```
interface MyThisInterface {
    m1: string[];
    m2: number[];
    functionA(): () => string;
}

let vMyThisInterface: MyThisInterface = {
    m1: ["hearts", "spades", "clubs", "diamonds"],
```

```
    m2: [1, 2, 3],
    functionA: function() {
        return function() {

            return this.m1[0]; // This is any
        };
    },
};

vMyThisInterface.functionA();
```



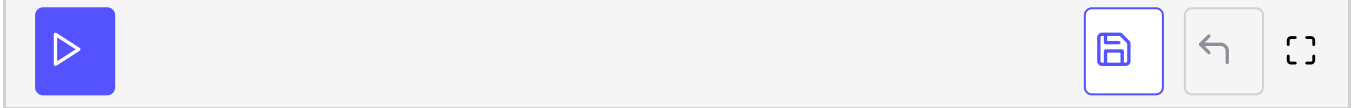
Defining a type to `this`

The `this` can be changed to explicitly specify the type to `this`. At **line 11**, the `this` is defined to be `MyInterface`. Hence, **line 12** the return type is as expected to be `string`.

```
interface MyThisInterface {
    m1: string[];
    m2: number[];
    functionA(): () => string;
}

let vMyThisInterface: MyThisInterface = {
    m1: ["hearts", "spades", "clubs", "diamonds"],
    m2: [1, 2, 3],
    functionA: function() {
        return function(this: MyThisInterface) {
            return this.m1[0];
        };
    },
};

vMyThisInterface.functionA();
```



You can also block the usage of `this` by using the same technique of specifying the type; however, this time using `void` instead of the type.

`this` with Callback

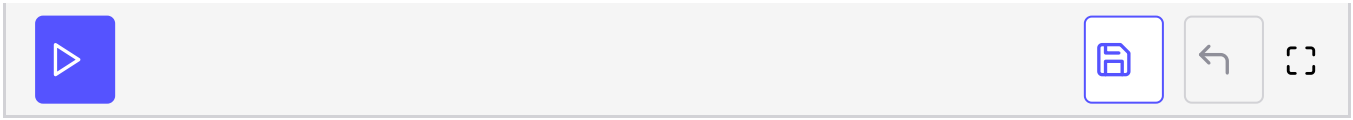
Often, the reference to `this` is lost when a function takes a callback function. By default, `this` refers to the context in which the function is called.

In the next example, **line** 7 has a `this` defined to `any` because it is inside an

(anonymous) callback function defined in **line 6**.

> 📜**Note:** the below code throws an error ✕

```
const family = {
    names: ["Patrick", "Alicia", "Melodie"],
    emotion: "love",
    print: function() {
        console.log("print", this); // this = the family object
        return this.names.forEach(function(name: string) {
            console.log("forEach", this); // this = implicit any = won't transpile
        });
    },
};
family.print();
```



In the example above, the first `console.log` prints the family object, but the second, inside the `forEach` function, is bound to the caller which is Node.js environment when executed from the command line or windows when executed from the browser.

Solving the callback `this`

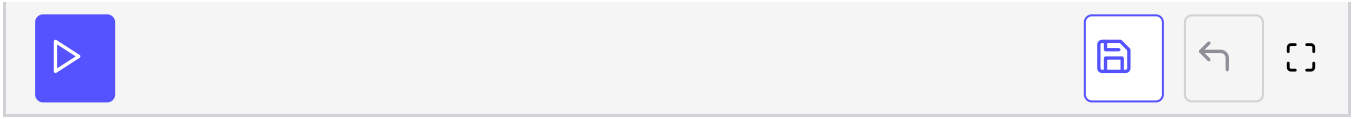
To solve this issue, using the fat arrow function will set this to the parent who is the family object. This solution is elegant because of its concise syntax. It also avoids setting the value of `this` which would be required if using other solutions available before **ECMAScript 2015** like using `bind` or passing `this` when the function allows setting the value of it by a parameter.

```
return this.names.forEach((name: string) => {});
return this.names.forEach(function(name: string) {}.bind(this));
return this.names.forEach(function(name: string) {}, family);
```

Here is the solution that passes a reference to `this` as well as giving `this` a proper type.

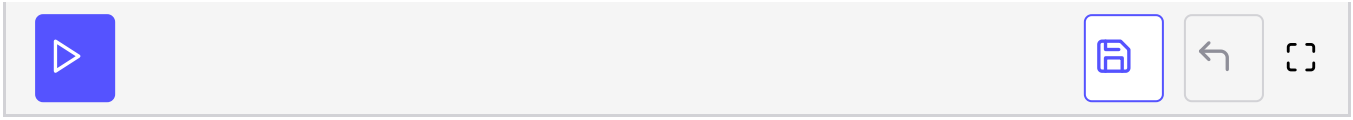
```
interface Family {
    names: string[];
    emotion: string;
    print: () => void;
}
const family: Family = {
```

```
    names: ["Patrick", "Alicia", "Melodie"],
    emotion: "love",
    print: function() {
        console.log("print", this); // this = the family object
        return this.names.forEach(function(this: Family, name: string) {
            console.log("forEach", this);
        }, family);
    },
};
family.print();
```



Here is the solution with the fat arrow which is simpler when `this` is already the right type.

```
    interface Family {
        names: string[];
        emotion: string;
        print: () => void;
    }
    const family: Family = {
        names: ["Patrick", "Alicia", "Melodie"],
        emotion: "love",
        print: function() {
            console.log("print", this); // this = the family object
            return this.names.forEach((name: string) => {
                console.log("forEach", this);
            });
        },
    };
    family.print();
```



Typing `this` is often not required and when cases occur, TypeScript lets you manually adjust the type for better type protection when needed. The capability of ensuring `this` to be well-typed can be critical to avoid errors.