# TRADITIONAL FUNCTIONS VS ARROW FUNCTIONS

**Sastry Kasibotla**

@Sastry-Kasibotla

01

# Traditional Functions vs Arrow Functions

Before ES6, we mostly used traditional function expressions. With ES6, a new type of function was introduced – arrow functions.

02

# The Syntax of Traditional Functions

The traditional function has a more verbose syntax.

```
function greet(name) {
    return "Hello, " + name + "!";
}
```

03

# Introducing Arrow Functions

Arrow functions provide a shorter syntax for writing functions.

```
const greet = (name) ⇒ {
    return "Hello, " + name + "!";
}
```

04

# Shortening Arrow Functions

For single parameter functions,
we can omit the parentheses.
For single line functions that immediately return, we can omit the curly braces and the return keyword.

```
const greet = name ⇒ "Hello, " + name + "!";
```

05

# "this" in Traditional vs Arrow Functions

Traditional functions bind their own this value, while arrow functions do not. Arrow functions capture the this value of the enclosing context.

# 05

## "this" in Traditional vs Arrow Functions

Traditional functions bind their own this value, while arrow functions do not. Arrow functions capture the this value of the enclosing context.

With traditional functions, **this** can be different based on how the function is called.

```
function Person() {
    this.age = 0;
    setInterval(function growUp() {
        this.age++;
    }, 1000);
}
var p = new Person();
```

In the above code, this inside growUp is not referring to the Person object but to the global context.

06

# Arrow Function "this" Example

Arrow functions capture the this value from their surroundings.

```
function Person() {
    this.age = 0;
    setInterval(() => {
        this.age++;
    }, 1000);
}
var p = new Person();
```

Now, this inside the arrow function refers to the Person object.

07

# When to Use Arrow Functions

## 1. Shorter Functions:

Arrow functions are ideal for situations where you need a concise expression without a lot of additional syntax. The brevity of arrow functions makes the code cleaner and more readable, especially when you're dealing with small, one-off functions.

```
// Traditional function
const square = function(x) {
    return x * x;
}

// Arrow function
const square = x => x * x;
```

Arrow functions are ideal for situations where you need a concise expression without a lot of additional syntax. The brevity of arrow functions makes the code cleaner and more readable, especially when you're dealing with small, one-off functions.

## 2. When You Need to Use this from the Surrounding Context:

Arrow functions don't have their own this binding. Instead, they inherit this from the surrounding lexical context. This makes them ideal for scenarios where you want to retain the context of the outer function.

```
function Timer() {
    this.seconds = 0;
    setInterval(() => {
        this.seconds++;
        console.log(this.seconds);
    }, 1000);
}
const timer = new Timer();
```

In the example above, the arrow function inside setInterval uses the this value of the Timer function. This behavior would be different with a traditional function.

## 3.Callbacks and Array Methods:

JavaScript developers often use higher-order functions like map, filter, and forEach. Arrow functions make these operations more concise.

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map(num => num * 2);
```

In the above example, the arrow function provides a short and clear way to double each number in the array.

08

# When Not to Use Arrow Functions

## 1. Methods Inside Classes (due to this behavior):

Arrow functions capture the this value of the enclosing context. When used as methods inside classes, they don't behave as one might expect, especially if you're coming from other object-oriented languages. Traditional function expressions are more predictable in this scenario.

```
class Person {
    constructor(name) {
        this.name = name;
    }

    // Using an arrow function as a method
    greet = () => {
        console.log(`Hello, ${this.name}`);
    }
}

const john = new Person('John');
const greet = john.greet;
greet();  // This will work because of the arrow function, but it's not a
typical behavior for class methods.
```

In the example, the arrow function captures the this value from the class, which can lead to unexpected behaviors in certain scenarios.

## 2. When You Need a Named Function for Debugging:

Named functions are beneficial during debugging because they provide a clear name in stack traces. Arrow functions are anonymous by nature, which can make debugging more challenging.

```
TypeError: ...
    at Object.<anonymous> ...
    at namedFunction ...
```

The named function (namedFunction) provides a clear reference point, while the arrow function appears as <anonymous>.

## 3. When You Want to Use the arguments Object:

Arrow functions don't have their own arguments object. They inherit it from the surrounding function. If you need to access the arguments object from a function, you should use a traditional function expression.

```
function traditionalFunction() {
    console.log(arguments);
}

const arrowFunction = () => {
    console.log(arguments);  // Will throw an error if not inside another
function.
}
```

In the above example, traditionalFunction can access the arguments object, while arrowFunction cannot unless it's nested inside another traditional function.

🙏 A Heartfelt Thank You! 🙏
Your support and engagement mean the world to me! If you found this content valuable, please give it a 👍 like and share it with your network.

🚀 For more insightful tips on Fintech, insights into the Startup world, and discussions on truly transformative topics, don't forget to hit that follow button!

Your support fuels this mission. 🌟 Stay curious, and let's keep the conversation going!

**Sastry Kasibotla**

@Sastry-Kasibotla