JavaScript Arrow Function

Arrow function is one of the features introduced in the ES6 version of JavaScript. It allows you to create functions in a cleaner way compared to regular functions. For example,

This function

```
// function expression
let x = function(x, y) {
   return x * y;
}
```

can be written as

```
// using arrow functions
let x = (x, y) => x * y;
```

using an arrow function.

Arrow Function Syntax

The syntax of the arrow function is:

```
let myFunction = (arg1, arg2, ...argN) => {
    statement(s)
}
```

Here,

- myFunction is the name of the function
- arg1, arg2, ...argN are the function arguments
- statement(s) is the function body

If the body has single statement or expression, you can write arrow function as:

let myFunction = (arg1, arg2, ...argN) => expression

Example 1: Arrow Function with No Argument

If a function doesn't take any argument, then you should use empty parentheses. For example,

```
let greet = () => console.log('Hello');
greet(); // Hello
```

Example 2: Arrow Function with One Argument

If a function has only one argument, you can omit the parentheses. For example,

```
let greet = x => console.log(x);
greet('Hello'); // Hello
```

Example 3: Arrow Function as an Expression

You can also dynamically create a function and use it as an expression. For example,

```
let age = 5;

let welcome = (age < 18) ?
  () => console.log('Baby') :
  () => console.log('Adult');

welcome(); // Baby
```

Example 4: Multiline Arrow Functions

If a function body has multiple statements, you need to put them inside curly brackets {}. For example,

```
let sum = (a, b) => {
    let result = a + b;
    return result;
}

let result1 = sum(5,7);
console.log(result1); // 12
```

this with Arrow Function

Inside a regular function, this keyword refers to the function where it is called.
However, this is not associated with arrow functions. Arrow function does not have its own this.
So whenever you call this, it refers to its parent scope. For example,

```
function Person() {
    this.name = 'Jack',
    this.age = 25,
    this.sayName = function () {
        // this is accessible
        console.log(this.age);
        function innerFunc() {
            // this refers to the global object
            console.log(this.age);
            console.log(this);
        }
        innerFunc();
    }
}
let x = new Person();
x.sayName();
```

Output

```
25
undefined
Window {}
```

Inside an arrow function

```
function Person() {
    this.name = 'Jack',
    this.age = 25,
    this.sayName = function () {
        console.log(this.age);
        let innerFunc = () => {
            console.log(this.age);
        }
 innerFunc();
    }
}
const x = new Person();
x.sayName();
```

Output

```
25
25
```

Here, the innerFunc() function is defined using the arrow function. And inside the arrow function, this refers to the parent's scope. Hence, this.age gives **25**.

Arguments Binding

Regular functions have arguments binding. That's why when you pass arguments to a regular function, you can access them using the arguments keyword. For example,

```
let x = function () {
    console.log(arguments);
}
x(4,6,7); // Arguments [4, 6, 7]
```

Arrow functions do not have arguments binding.
When you try to access an argument using the arrow function, it will give an error. For example,

```
let x = () => {
    console.log(arguments);
}
x(4,6,7);
// ReferenceError: Can't find variable: arguments
```

Arrow Function with Promises and Callbacks

Arrow functions provide better syntax to write promises and callbacks. For example,

```
// ES5
asyncFunction().then(function() {
    return asyncFunction1();
}).then(function() {
    return asyncFunction2();
}).then(function() {
    finish;
});
```

can be written as

```
// ES6
asyncFunction()
.then(() => asyncFunction1())
.then(() => asyncFunction2())
.then(() => finish);
```

JavaScript Default Parameters

The concept of default parameters is a new feature introduced in the ES6 version of JavaScript. This allows us to give default values to function parameters. Let's take an example,

```
function sum(x = 3, y = 5) {

    // return sum
    return x + y;
}

console.log(sum(5, 15));  // 20
console.log(sum(7));      // 12
console.log(sum());       // 8
```

In the above example, the default value of x is **3** and the default value of y is **5**.

- sum(5, 15) - When both arguments are passed, x takes **5** and y takes **15**.
- sum(7) - When **7** is passed to the sum() function, x takes **7** and y takes default value **5**.
- sum() - When no argument is passed to the **sum()** function, x takes default value **3** and y takes default value **5**.

JavaScript Template Literals (Template Strings)

Template literals (template strings) allow you to use strings or embedded expressions in the form of a string. They are enclosed in backticks ``. For example,

```
const name = 'Jack';
console.log(`Hello ${name}!`); // Hello Jack!
```

Template Literals for Strings

In the earlier versions of JavaScript, you would use a single quote '' or a double quote "" for strings. For example,

```
const str1 = 'This is a string';
// cannot use the same quotes
const str2 = 'A "quote" inside a string';  // valid code
const str3 = 'A 'quote' inside a string';  // Error

const str4 = "Another 'quote' inside a string"; // valid code
const str5 = "Another "quote" inside a string"; // Error
```