JavaScript TUTORIAL

JavaScript Arrow Functions

Summary: in this tutorial, you will learn how to use the JavaScript arrow function to write more concise code for function expressions.

Introduction to JavaScript arrow functions

ES6 arrow functions provide an alternative way to write a shorter syntax compared to the function expression.

The following example defines a function expression that returns the sum of two numbers:

```
let add = function (x, y) {
  return x + y;
};

console.log(add(10, 20)); // 30
```

The following example is equivalent to the above `add()` function expression but use an arrow function instead:

```
let add = (x, y) => x + y;

console.log(add(10, 20)); // 30;
```

In this example, the arrow function has one expression `x + y` so it returns the result of the expression.

However, if you use the block syntax, you need to explicitly use the `return` keyword:

```
let add = (x, y) => {
  return x + y;
};
```

The `typeof` operator returns `function` indicating the type of arrow function.

```
console.log(typeof add); // function
```

The arrow function is also an instance of the Function type as shown in the following example:

```
console.log(add instanceof Function); // true
```

JavaScript arrow functions with multiple parameters

If an arrow function has two or more parameters, you use the following syntax:

```
(p1, p2, ..., pn) => expression;
```

The following expression:

```
=> expression
```

is equivalent to the following expression:

```
=> { return expression; }
```

For example, to sort an array of numbers in the descending order, you use the `sort()` method of the array object as follows:

```
let numbers = [4, 2, 6];

numbers.sort(function (a, b) {
  return b - a;
});
console.log(numbers); // [6,4,2]
```

The code is more concise with the arrow function syntax:

```
let numbers = [4, 2, 6];
numbers.sort((a, b) => b - a);

console.log(numbers); // [6,4,2]
```

JavaScript arrow functions with a single parameter

If an arrow function takes a single parameter, you use the following syntax:

```
(p1) => { statements }
```

Note that you can omit the parentheses as follows:

```
p => { statements }
```

The following example uses an arrow function as an argument of the `map()` method that transforms an array of strings into an array of the string's lengths.

```
let names = ['John', 'Mac', 'Peter'];
let lengths = names.map(name => name.length);

console.log(lengths);
```

Output:

```
[ 4, 3, 5 ]
```

JavaScript arrow functions with no parameter

If the arrow function has no parameter, you need to use parentheses, like this:

```
() => { statements }
```

For example:

```
let logDoc = () => console.log(window.document);
logDoc();
```

Line break between parameter definition and arrow

JavaScript doesn't allow a line break between the parameter definition and the arrow (`=>`) in an arrow function.

For example, the following code causes a `SyntaxError` :

```
let multiply = (x,y)
=> x * y;
```

However, the following code works perfectly fine:

```
let multiply = (x,y) =>
x * y;
```

JavaScript allows you to have line breaks between parameters as shown in the following example:

```
let multiply = (
  x,
  y
) =>
x * y;
```

Statements & expressions in the arrow function body

In JavaScript, an expression evaluates to a value as shown in the following example.

```
10 + 20;
```

A statement does a specific task such as:

```
if (x === y) {
    console.log('x equals y');
}
```

If you use an expression in the body of an arrow function, you don't need to use the curly braces.

```
let square = x => x * x;
```

However, if you use a statement, you must wrap it inside a pair of curly braces as in the following example:

```
let except = msg => {
    throw msg;
};
```

JavaScript arrow functions and object literals

Consider the following example:

```
let setColor = function (color) {
  return { value: color };
};

let backgroundColor = setColor('Red');
console.log(backgroundColor.value); // "Red"
```

The `setColor()` function expression returns an object that has the `value` property set to the `color` argument.

If you use the following syntax to return an object literal from an arrow function, you will get an error.

```
p => {object:literal}
```

For example, the following code causes an error.

```
let setColor = color => {value: color };
```

Since both block and object literal use curly brackets, the JavasScript engine cannot distinguish between a block and an object.

To fix this, you need to wrap the object literal in parentheses as follows:

```
let setColor = color => ({value: color });
```

Arrow function vs. regular function

There are two main differences between an arrow function and a regular function.

1. First, in the arrow function, the `this`, `arguments`, `super`, `new.target` are lexical. It means that the arrow function uses these variables (or constructs) from the enclosing lexical scope.
2. Second, an arrow function cannot be used as a function constructor. If you use the `new` keyword to create a new object from an arrow function, you will get an error.

JavaScript arrow functions and this value

In JavaScript, a new function defines its own `this` value. However, this is not the case for the arrow function. See the following example:

```
function Car() {
  this.speed = 0;

  this.speedUp = function (speed) {
    this.speed = speed;
    setTimeout(function () {
      console.log(this.speed); // undefined
    }, 1000);
  };
}

let car = new Car();
car.speedUp(50);
```

Inside the anonymous function of the `setTimeout()` function, the `this.speed` is `undefined`. The reason is that the `this` of the anonymous function shadows the `this` of the `speedUp()` method.

To fix this, you assign the `this` value to a variable that doesn't shadow inside the anonymous function as follows:

```
function Car() {
  this.speed = 0;

  this.speedUp = function (speed) {
    this.speed = speed;
    let self = this;
    setTimeout(function () {
      console.log(self.speed);
    }, 1000);
  };
}

let car = new Car();
car.speedUp(50); // 50;
```

Unlike an anonymous function, an arrow function captures the `this` value of the enclosing context instead of creating its own `this` context. The following code should work as expected:

```
function Car() {
  this.speed = 0;

  this.speedUp = function (speed) {
    this.speed = speed;
    setTimeout(() => console.log(this.speed), 1000);
  };
}

let car = new Car();
car.speedUp(50); // 50;
```

JavaScript arrow functions and the arguments object

An arrow function doesn't have the `arguments` object. For example:

```
function show() {
  return (x) => x + arguments[0];
}

let display = show(10, 20);
let result = display(5);
console.log(result); // 15
```

The arrow function inside the `show()` function references the `arguments` object. However, this `arguments` object belongs to the `show()` function, not the arrow function.

An arrow function also doesn't have the `new.target` keyword.

JavaScript arrow functions and the prototype property

When you define a function using a `function` keyword, the function has a property called `prototype`:

```
function dump(message) {
  console.log(message);
}
console.log(dump.hasOwnProperty('prototype')); // true
```

However, arrow functions don't have the `prototype` property:

```
let dump = message => console.log(message);
console.log(dump.hasOwnProperty('prototype')); // false
```

It is a good practice to use arrow functions for callbacks and closures because the syntax of arrow functions is cleaner.

Summary

- Use the `(...args) => expression;` to define an arrow function.
- Use the `(...args) => { statements }` to define an arrow function that has multiple statements.
- An arrow function doesn't have its binding to `this` or `super`.
- An arrow function doesn't have `arguments` object, `new.target` keyword, and `prototype` property.

Quiz