JS

ECMA Script 2015

#JavaScript Notes

ECMA Script 2015
ES6

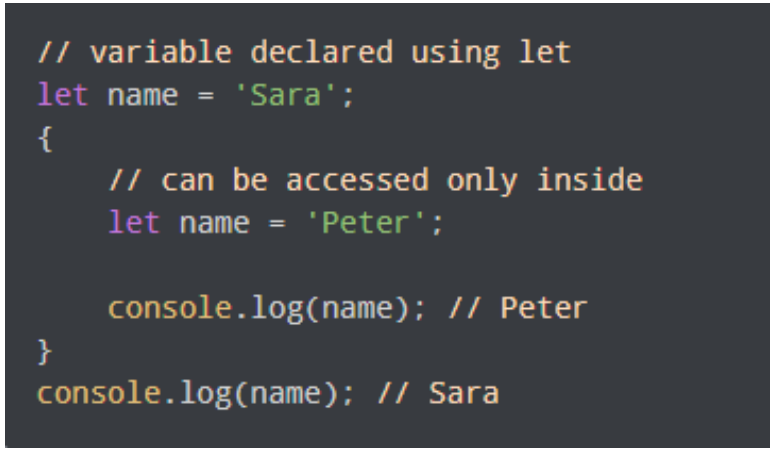
JS

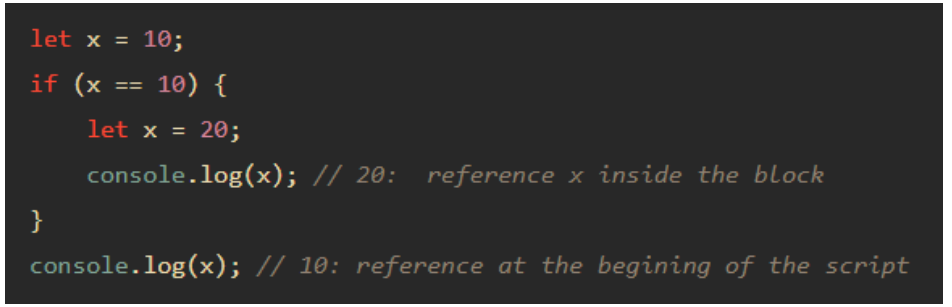
Let,Const

JavaScript let

- JavaScript let is used to declare variables. Previously, variables were declared using the var keyword.
- The variables declared using let are block-scoped. This means they are only accessible within a particular block. For example,

```
// variable declared using let
let name = 'Sara';
{
   // can be accessed only inside
   let name = 'Peter';

   console.log(name); // Peter
}
console.log(name); // Sara
```





JavaScript const

- The const statement is used to declare constants in JavaScript.
- you cannot change the value of a const variable.
- Syntax : const CONSTANT_NAME = value;

```
// name declared with const cannot be changed
const name = 'Sara';
```


```
// name declared with const cannot be changed
const name = 'Sara';
```

JavaScript let and global object

- When you declare a global variable using the var keyword, you add that variable to the property list of the global object. In the case of the web browser, the global object is the window. For example:

```
var a = 10;
console.log(window.a); // 10
```

- However, when you use the let keyword to declare a variable, that variable is not attached to the global object as a property. For example:

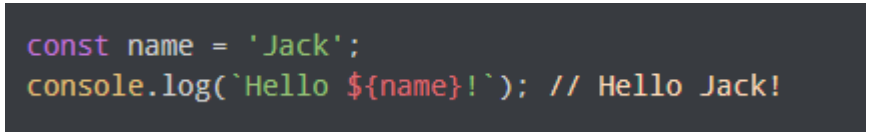
```
let b = 20;
console.log(window.b); // undefined
```

JavaScript Template Literals (Template Strings)

JS

- 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,

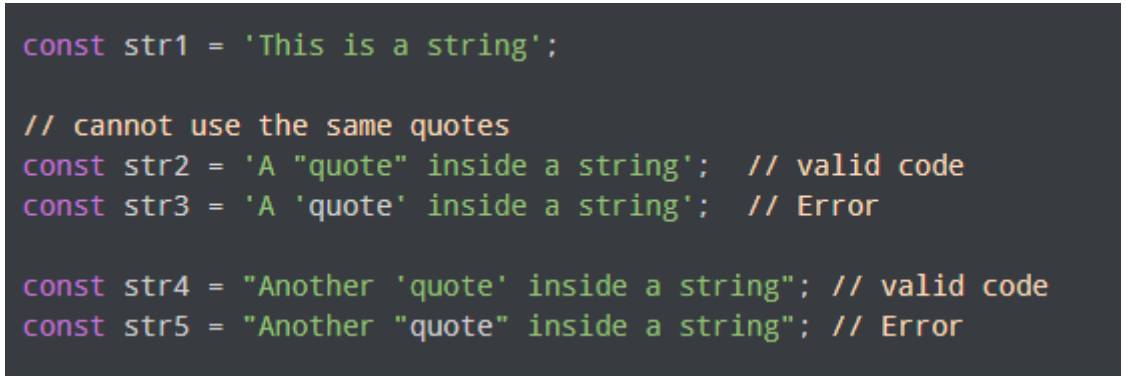
```
const name = 'Jack';
console.log(`Hello ${name}!`); // Hello Jack!
```



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
```



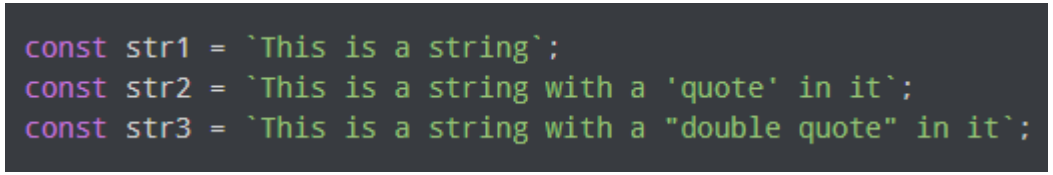
- To use the same quotations inside the string, you can use the escape character \.

```
// escape characters using \
const str3 = 'A \'quote\' inside a string';  // valid code
const str5 = "Another \"quote\" inside a string"; // valid
code
```

```
// escape characters using \
const str3 = 'A \'quote\' inside a string';  // valid code
const str5 = "Another \"quote\" inside a string"; // valid code
```

- Instead of using escape characters, you can use template literals. For example,

```
const str1 = `This is a string`;
const str2 = `This is a string with a 'quote' in it`;
const str3 = `This is a string with a "double quote" in it`;
```



- As you can see, the template literals not only make it easy to include quotations but also make our code look cleaner.

Multiline Strings Using Template Literals

- Template literals also make it easy to write multiline strings. For example,

- Using template literals, you can replace

```
// using the + operator
const message1 = 'This is a long message\n' +
'that spans across multiple lines\n' +
'in the code.'

console.log(message1)
```

```
// using the + operator
const message1 = 'This is a long message\n' +
'that spans across multiple lines\n' +
'in the code.'

console.log(message1)
```

- With

```
const message1 = `This is a long message
that spans across multiple lines
in the code.`

console.log(message1)
```

```
const message1 = `This is a long message
that spans across multiple lines
in the code.`

console.log(message1)
```

- The output of both these programs will be the same.

```
This is a long message
that spans across multiple lines
in the code.
```

Expression Interpolation

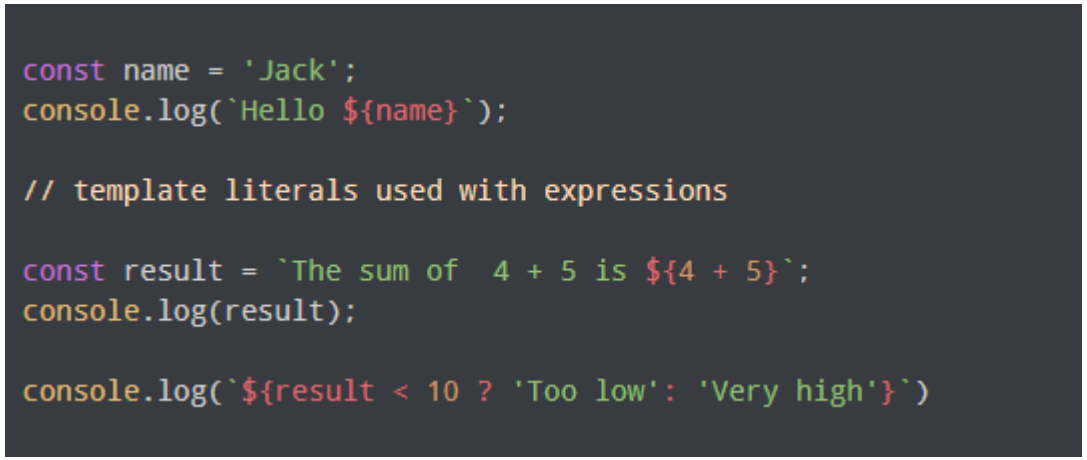
- Before JavaScript ES6, you would use the + operator to concatenate variables and expressions in a string. For example,

```
const name = 'Jack';
console.log('Hello ' + name); // Hello Jack
```


```
const name = 'Jack';
console.log('Hello ' + name); // Hello Jack
```

- With template literals, it's a bit easier to include variables and expressions inside a string. For that, we use the ${...} syntax.

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

// template literals used with expressions

const result = `The sum of  4 + 5 is ${4 + 5}`;
console.log(result);

console.log(`${result < 10 ? 'Too low': 'Very high'}`)
```



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

// template literals used with expressions

const result = `The sum of  4 + 5 is ${4 + 5}`;
console.log(result);

console.log(`${result < 10 ? 'Too low': 'Very high'}`)
```

Output

```
Hello Jack
The sum of  4 + 5 is 9
Very high
```

Tagged Templates

- Normally, you would use a function to pass arguments. For example,

```
function tagExample(strings) {
    return strings;
}

// passing argument
const result = tagExample('Hello Jack');

console.log(result);
```

```
function tagExample(strings) {
    return strings;
}

// passing argument
const result = tagExample('Hello Jack');

console.log(result);
```

Output

```
["Hello Jack"]
```

- An array of string values are passed as the first argument of a tag function. You could also pass the values and expressions as the remaining arguments. For example,

```
const name = 'Jack';
const greet = true;
function tagExample(strings, nameValue) {
    let str0 = strings[0]; // Hello
    let str1 = strings[1]; // , How are you?

    if(greet) {
        return `${str0}${nameValue}${str1}`;
    }
}
// creating tagged literal
// passing argument name
const result = tagExample`Hello ${name}, How are you?`;
console.log(result);
```

```
const name = 'Jack';
const greet = true;

function tagExample(strings, nameValue) {
    let str0 = strings[0]; // Hello
    let str1 = strings[1]; // , How are you?

    if(greet) {
        return `${str0}${nameValue}${str1}`;
    }
}

// creating tagged literal
// passing argument name
const result = tagExample`Hello ${name}, How are you?`;

console.log(result);
```

Output

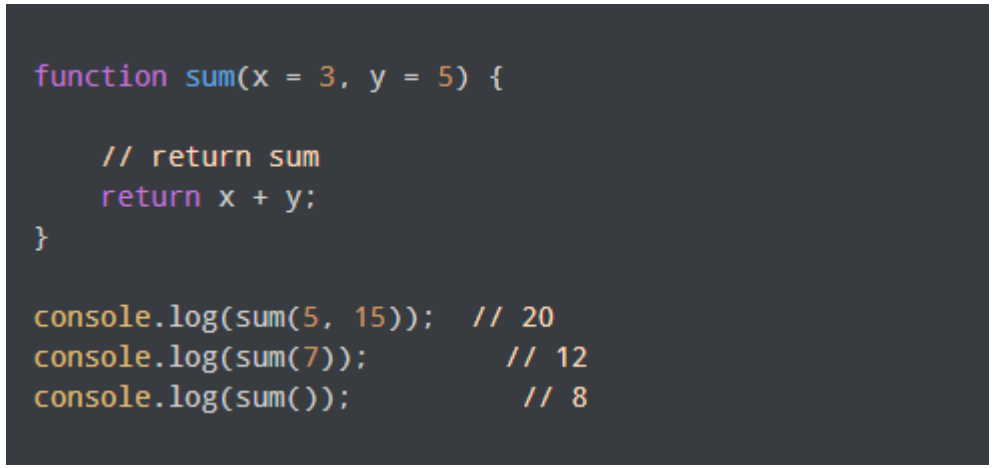
```
Hello Jack, How are you?
```

JavaScript Default Parameters

JS

- 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.

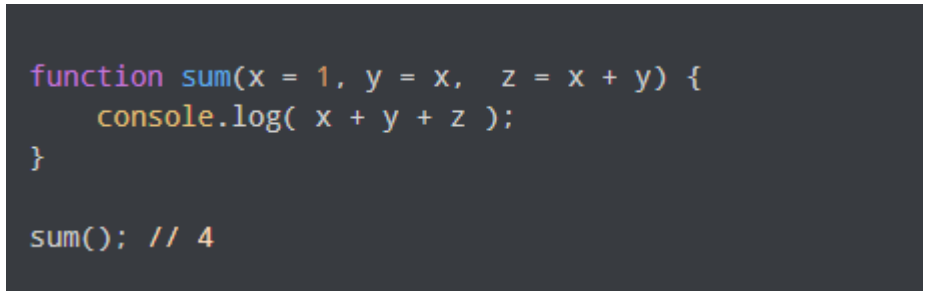
```
function sum(x = 3, y = 5) {

   // return sum
   return x + y;
}

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



Example 1: Passing Parameter as Default Values

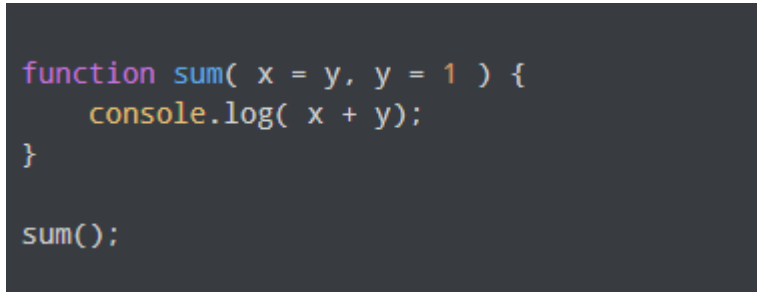
- Using Expressions as Default Values
- It is also possible to provide expressions as default values.

```
function sum(x = 1, y = x,  z = x + y) {
   console.log( x + y + z );
}

sum(); // 4
```



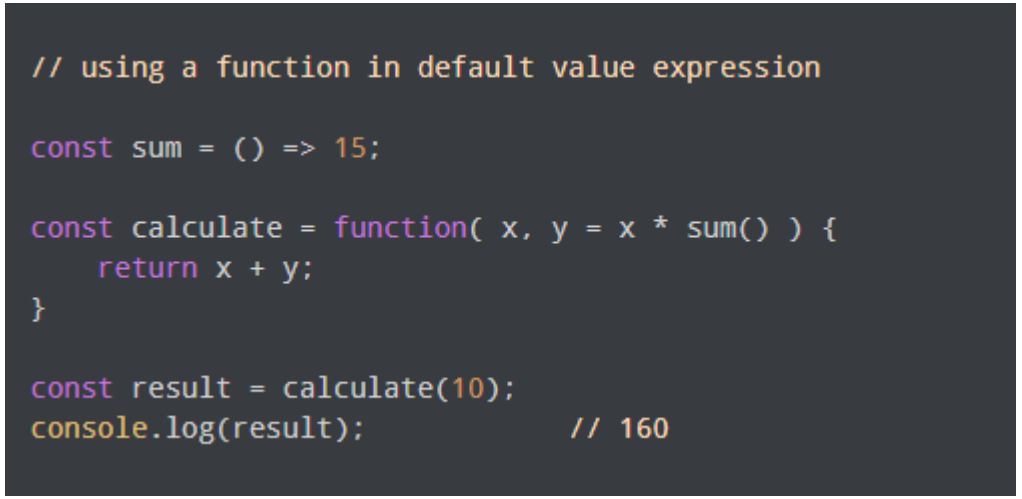
- If you reference the parameter that has not been initialized yet, you will get an error. For example,

```
function sum( x = y, y = 1 ) {
   console.log( x + y);
}

sum();
```



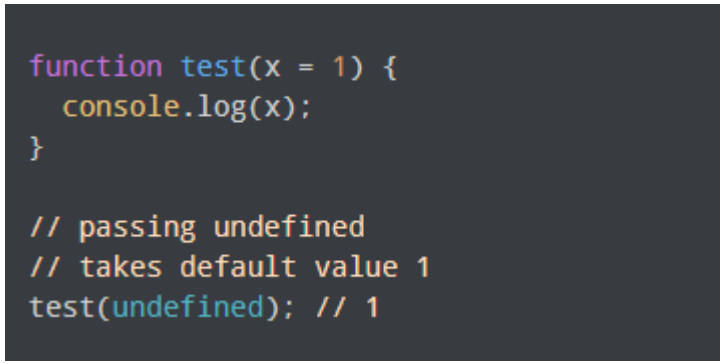
Example 2: Passing Function Value as Default Value

```
// using a function in default value expression

const sum = () => 15;

const calculate = function( x, y = x * sum() ) {
   return x + y;
}

const result = calculate(10);
console.log(result);          // 160
```



- Passing undefined Value
- In JavaScript, when you pass undefined to a default parameter function, the function takes the default value. For example,

```
function test(x = 1) {
  console.log(x);
}

// passing undefined
// takes default value 1
test(undefined); // 1
```



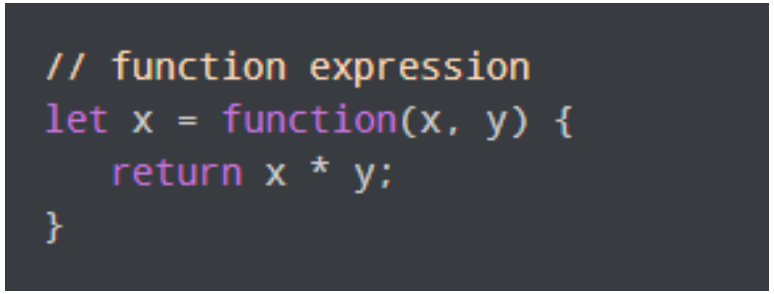
Aero

JavaScript Arrow Function

- In the ES6 version, you can use arrow functions to create function expressions. For example,

- This function

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



- can be written as

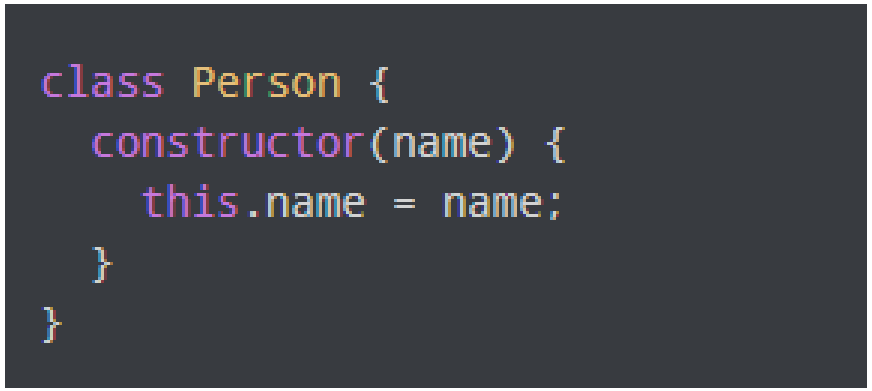
```
// function expression using arrow function
let x = (x, y) => x * y;
```


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

JavaScript Classes

- JavaScript class is used to create an object. Class is similar to a constructor function. For example,

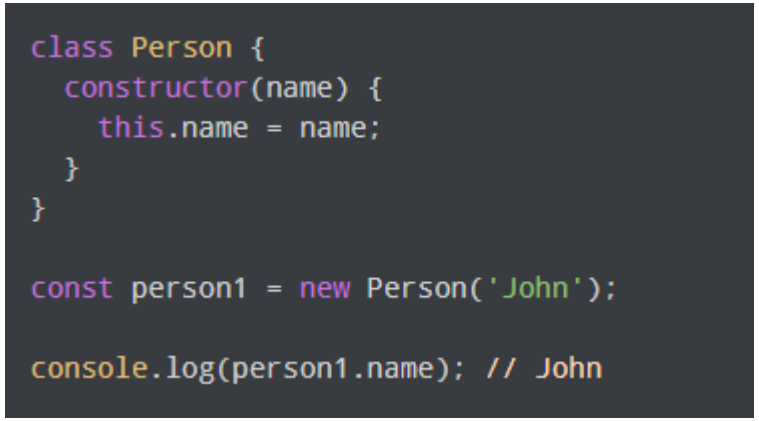
```
class Person {
  constructor(name) {
    this.name = name;
  }
}
```



- Keyword class is used to create a class. The properties are assigned in a constructor function.

- Now you can create an object. For example,

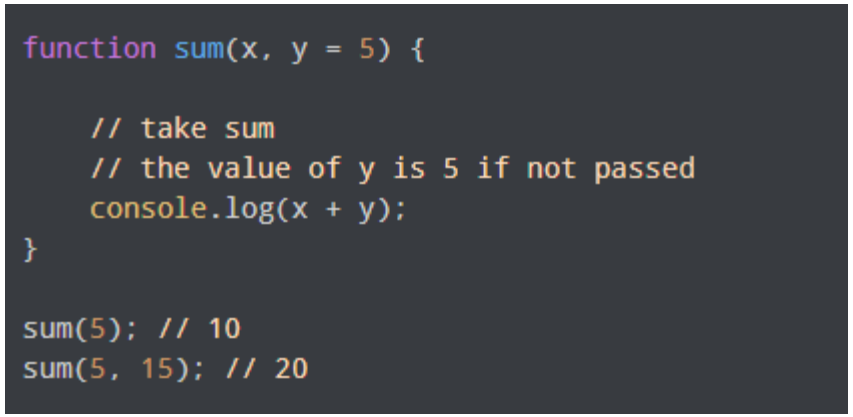
```
class Person {
  constructor(name) {
    this.name = name;
  }
}

const person1 = new Person('John');

console.log(person1.name); // John
```



Default Parameter Values

- In the ES6 version, you can pass default values in the function parameters. For example,

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

    // take sum
    // the value of y is 5 if not passed
    console.log(x + y);
}

sum(5); // 10
sum(5, 15); // 20
```



JavaScript Template Literals

- The template literal has made it easier to include variables inside a string. For example, before you had to do:

```
const first_name = "Jack";
const last_name = "Sparrow";

console.log('Hello ' + first_name + ' ' + last_name);
```

```
const first_name = "Jack";
const last_name = "Sparrow";

console.log('Hello ' + first_name + ' ' + last_name);
```

- This can be achieved using template literal by:

```
const first_name = "Jack";
const last_name = "Sparrow";

console.log(`Hello ${first_name} ${last_name}`);
```


```
const first_name = "Jack";
const last_name = "Sparrow";

console.log(`Hello ${first_name} ${last_name}`);
```

JavaScript Destructuring

- The destructuring syntax makes it easier to assign values to a new variable. For example,

```
// before you would do something like this
const person = {
    name: 'Sara',
    age: 25,
    gender: 'female'
}

let name = person.name;
let age = person.age;
let gender = person.gender;

console.log(name); // Sara
console.log(age); // 25
console.log(gender); // female
```


```
// before you would do something like this
const person = {
    name: 'Sara',
    age: 25,
    gender: 'female'
}

let name = person.name;
let age = person.age;
let gender = person.gender;

console.log(name); // Sara
console.log(age); // 25
console.log(gender); // female
```

- Using ES6 Destructuring syntax, the above code can be written as:

```
const person = {
    name: 'Sara',
    age: 25,
    gender: 'female'
}

let { name, age, gender } = person;

console.log(name); // Sara
console.log(age); // 25
console.log(gender); // female
```

JavaScript import and export

- You could export a function or a program and use it in another program by importing it. This helps to make reusable components. For example, if you have two JavaScript files named contact.js and home.js.

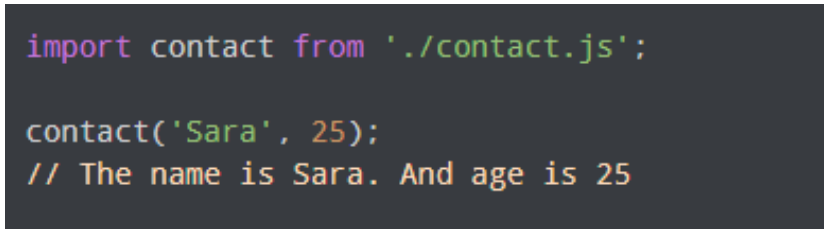
- In contact.js file, you can export the contact() function:

```
// export
export default function contact(name, age) {
   console.log(`The name is ${name}. And age is
${age}.`);
}
```

```
// export
export default function contact(name, age) {
    console.log(`The name is ${name}. And age is ${age}.`);
}
```

- Then when you want to use the contact() function in another file, you can simply import the function. For example, in home.js file:

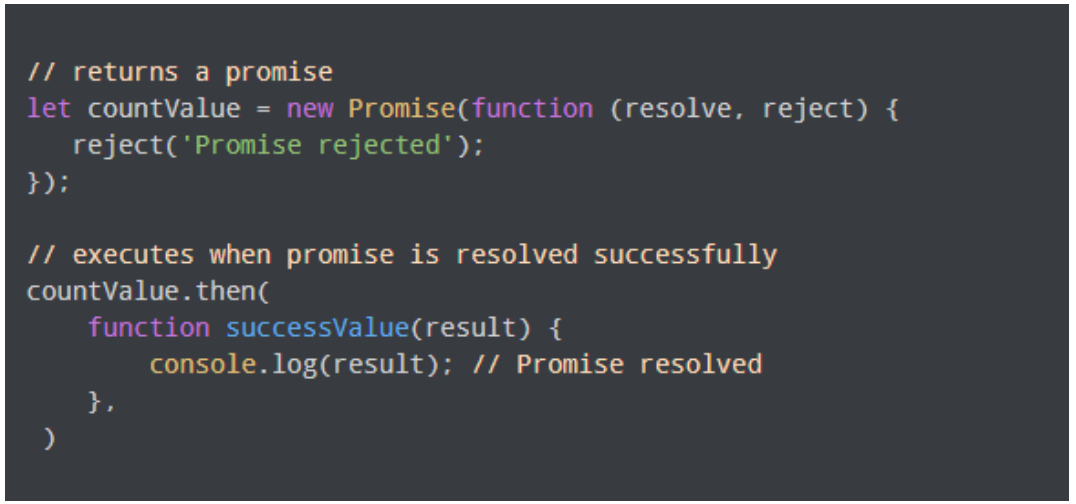
```
import contact from './contact.js';

contact('Sara', 25);
// The name is Sara. And age is 25
```



JavaScript Promises

- Promises are used to handle asynchronous tasks. For example,

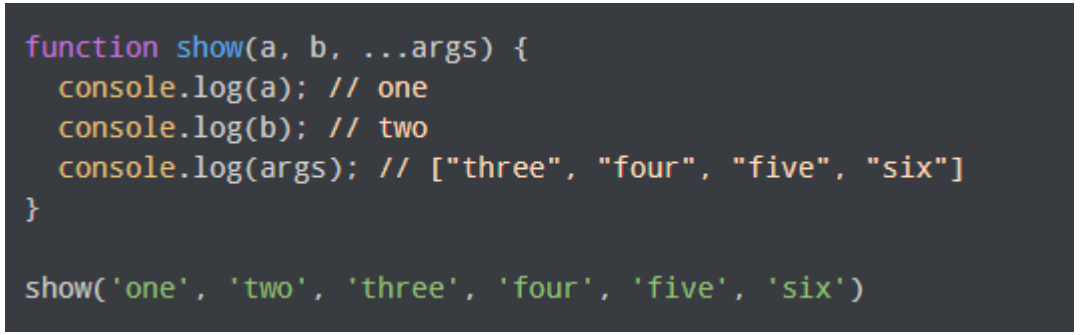
```
// returns a promise
let countValue = new Promise(function (resolve, reject) {
   reject('Promise rejected');
});

// executes when promise is resolved successfully
countValue.then(
    function successValue(result) {
        console.log(result); // Promise resolved
    },
 )
```



JavaScript Rest Parameter and Spread Operator

- You can use the rest parameter to represent an indefinite number of arguments as an array. For example,

```
function show(a, b, ...args) {
  console.log(a); // one
  console.log(b); // two
  console.log(args); // ["three", "four", "five", "six"]
}

show('one', 'two', 'three', 'four', 'five', 'six')
```



- You pass the remaining arguments using ... syntax. Hence, the name rest parameter.

- You use the spread syntax ... to copy the items into a single array. For example,

```
let arr1 = ['one', 'two'];
let arr2 = [...arr1, 'three', 'four', 'five'];
console.log(arr2); // ["one", "two", "three", "four", "five"]
```


```
let arr1 = ['one', 'two'];
let arr2 = [...arr1, 'three', 'four', 'five'];
console.log(arr2); // ["one", "two", "three", "four", "five"]
```

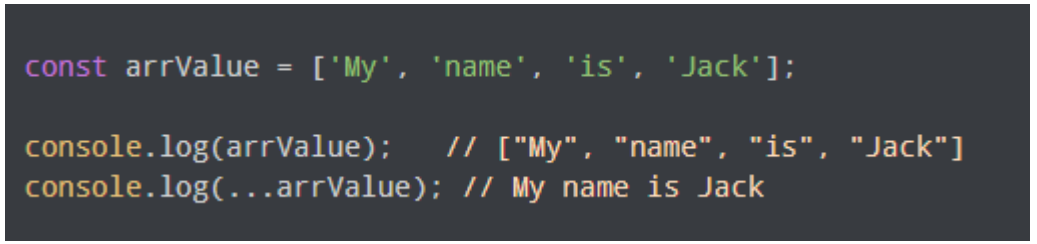
JavaScript Spread Operator

JS

Spread Operator

- The spread operator ... is used to expand or spread an iterable or an array. For example,

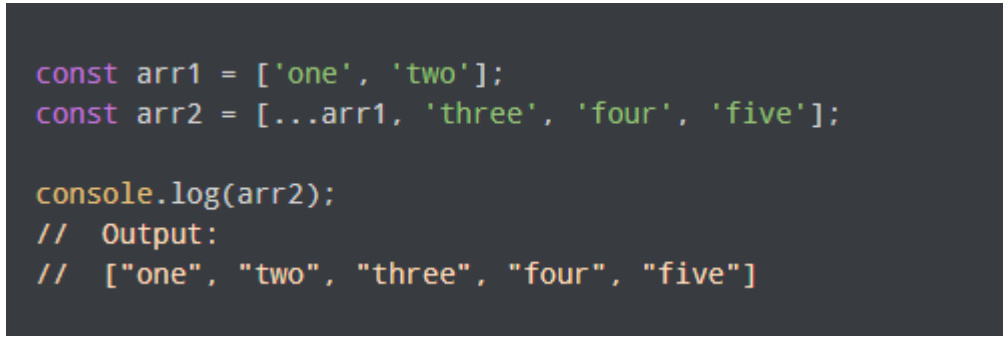
```
const arrValue = ['My', 'name', 'is', 'Jack'];

console.log(arrValue);  // ["My", "name", "is", "Jack"]
console.log(...arrValue); // My name is Jack
```



Copy Array Using Spread Operator

- You can also use the spread syntax ... to copy the items into a single array. For example,

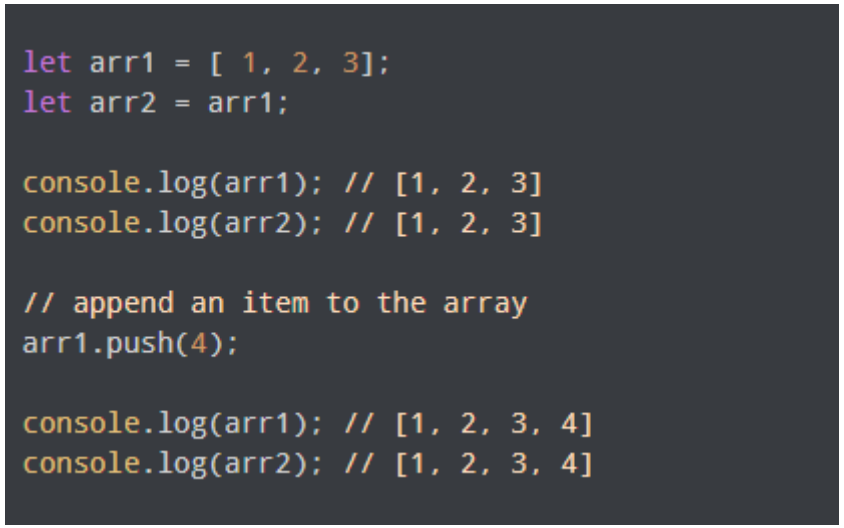
```
const arr1 = ['one', 'two'];
const arr2 = [...arr1, 'three', 'four', 'five'];

console.log(arr2);
//  Output:
//  ["one", "two", "three", "four", "five"]
```



Clone Array Using Spread Operator

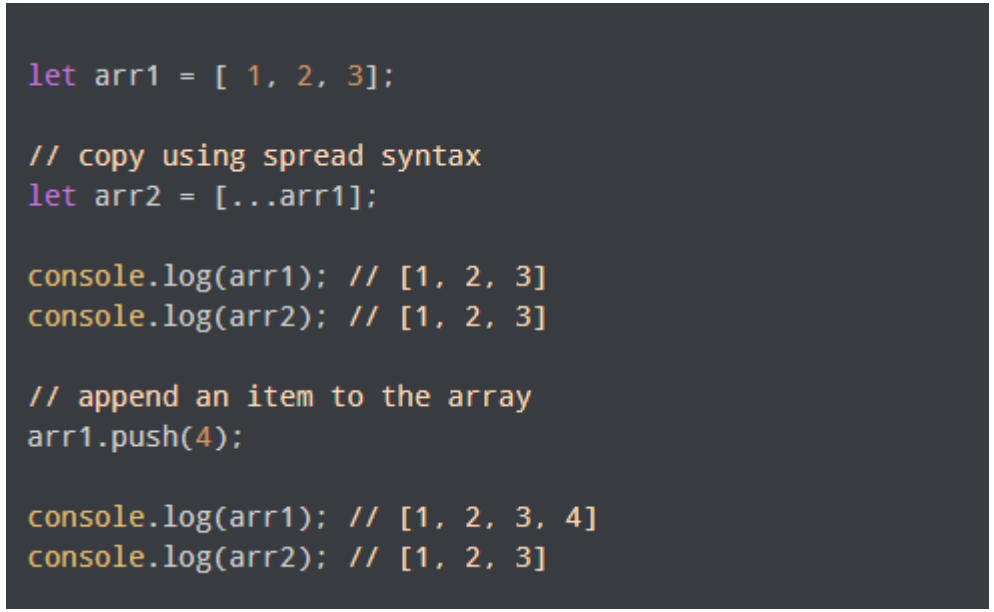
- In JavaScript, objects are assigned by reference and not by values. For example,

```
let arr1 = [ 1, 2, 3];
let arr2 = arr1;

console.log(arr1); // [1, 2, 3]
console.log(arr2); // [1, 2, 3]

// append an item to the array
arr1.push(4);

console.log(arr1); // [1, 2, 3, 4]
console.log(arr2); // [1, 2, 3, 4]
```



- However, if you want to copy arrays so that they do not refer to the same array, you can use the spread operator.

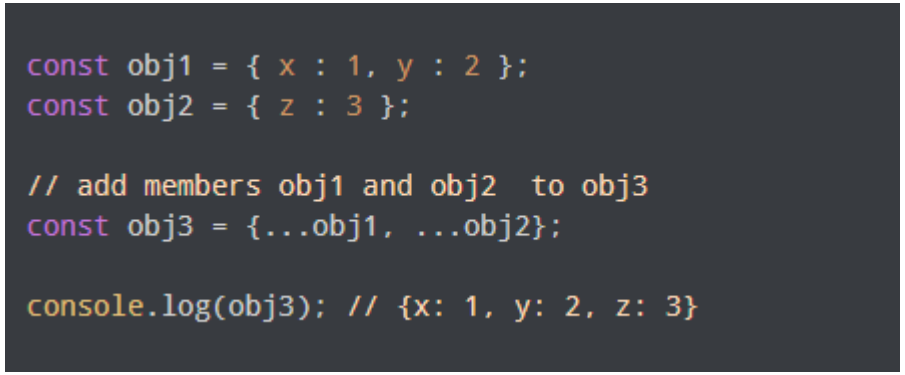
```
let arr1 = [ 1, 2, 3];

// copy using spread syntax
let arr2 = [...arr1];

console.log(arr1); // [1, 2, 3]
console.log(arr2); // [1, 2, 3]

// append an item to the array
arr1.push(4);

console.log(arr1); // [1, 2, 3, 4]
console.log(arr2); // [1, 2, 3]
```



Spread Operator with Object

- You can also use the spread operator with object literals. For example,

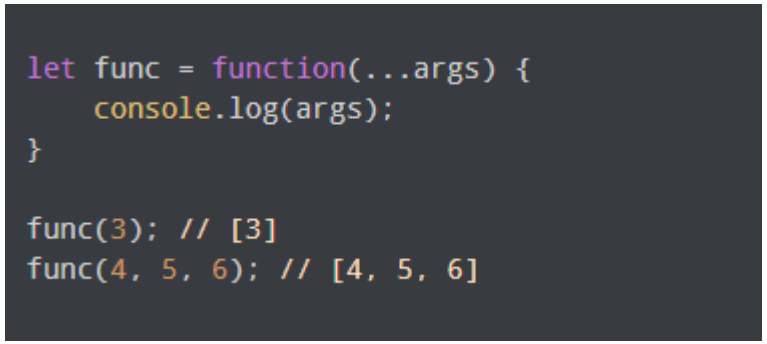
```
const obj1 = { x : 1, y : 2 };
const obj2 = { z : 3 };

// add members obj1 and obj2  to obj3
const obj3 = {...obj1, ...obj2};

console.log(obj3); // {x: 1, y: 2, z: 3}
```



Rest Parameter

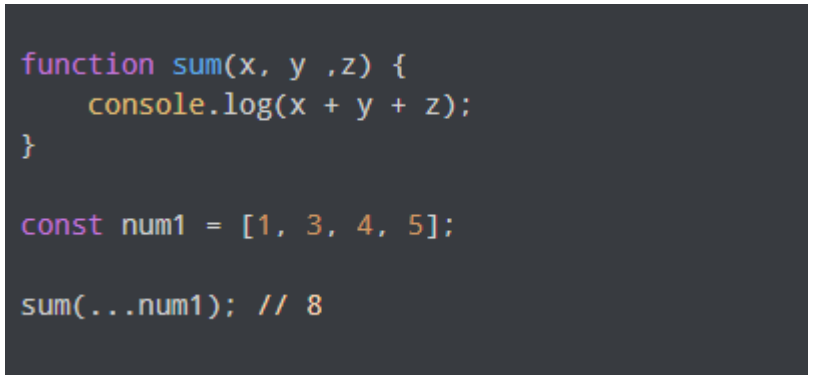
- When the spread operator is used as a parameter, it is known as the rest parameter.
- You can also accept multiple arguments in a function call using the rest parameter. For example,

```
let func = function(...args) {
    console.log(args);
}

func(3); // [3]
func(4, 5, 6); // [4, 5, 6]
```



- You can also pass multiple arguments to a function using the spread operator.
- If you pass multiple arguments using the spread operator, the function takes the required arguments and ignores the rest.

```
function sum(x, y ,z) {
   console.log(x + y + z);
}

const num1 = [1, 3, 4, 5];

sum(...num1); // 8
```

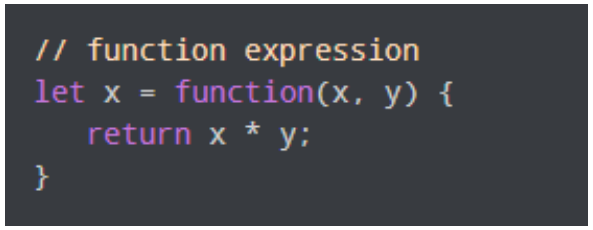


JavaScript Arrow Function

JS

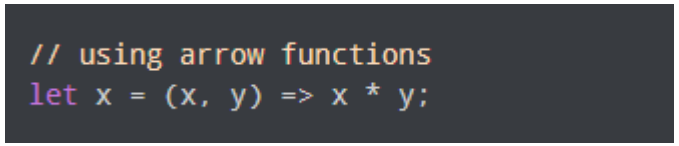
- Arrow function is one of the features introduced in the ES6 version of JavaScript. It allows you to create functions in a cleaner way compared to regular functions. For example,
- This function

```
// 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,

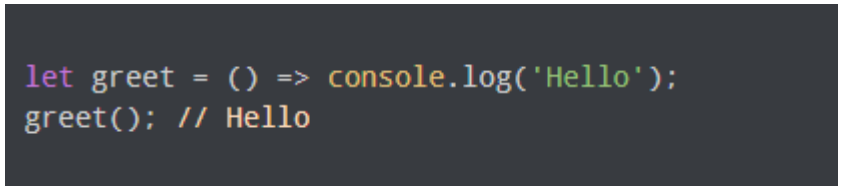
- myFunction is the name of the function
- arg1, arg2, ...argN are the function arguments
- statement(s) is the function body
- If the body has single statement or expression, you can write arrow function as:

```
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,

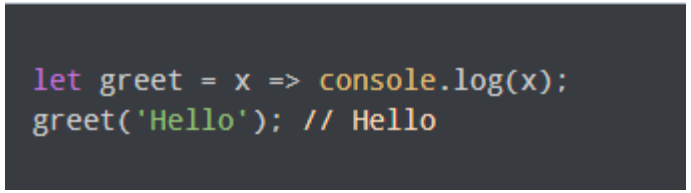
```
let greet = () => console.log('Hello');
greet(); // Hello
```



Example 2: Arrow Function with One Argument

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

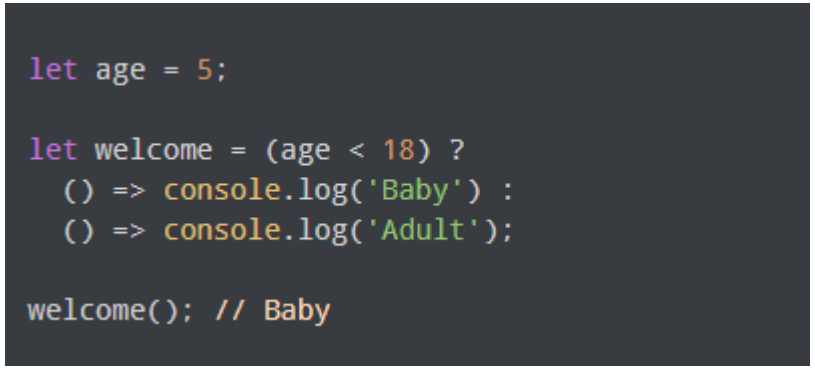
```
let greet = x => console.log(x);
greet('Hello'); // Hello
```



Example 3: Arrow Function as an Expression

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

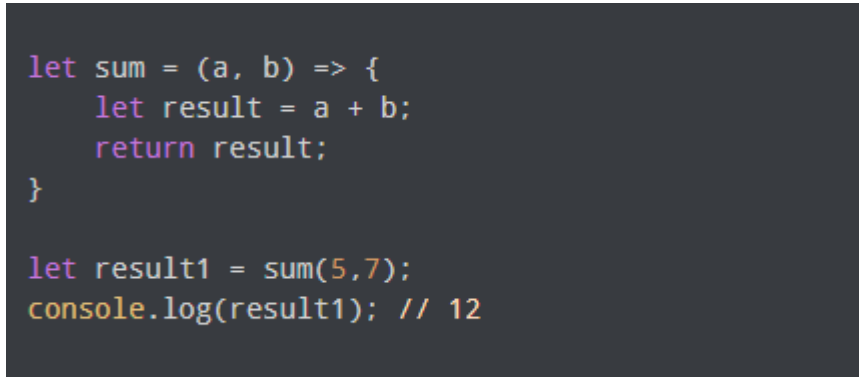
```
let age = 5;

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,

- Inside a regular function

```
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 {}
```

```
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();
```

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();
```

```
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();
```

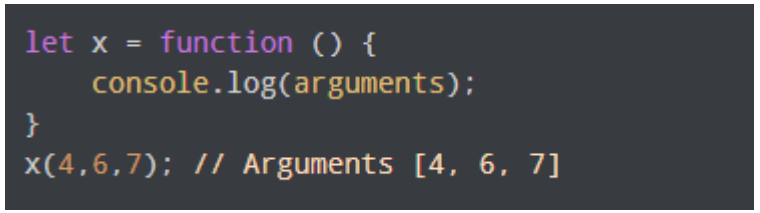
Output

```
25
25
```

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,

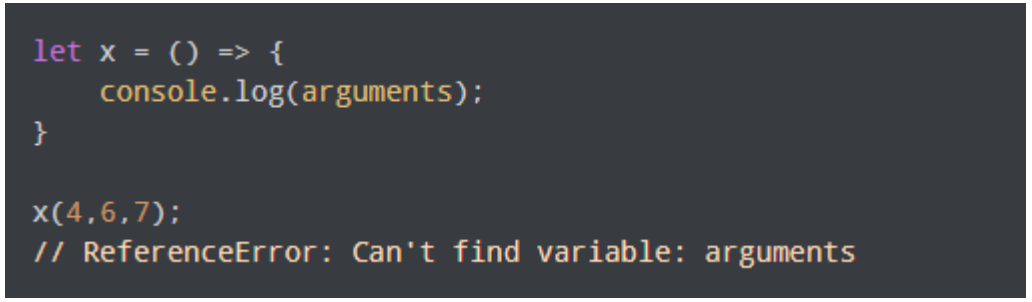
```
let x = function () {
    console.log(arguments);
}
x(4,6,7); // Arguments [4, 6, 7]
```



- Arrow functions do not have arguments binding.

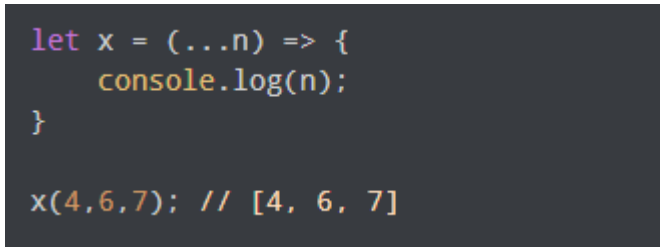
- When you try to access an argument using the arrow function, it will give an error. For example,

```
let x = () => {
    console.log(arguments);
}

x(4,6,7);
// ReferenceError: Can't find variable: arguments
```



- To solve this issue, you can use the spread syntax. For example,

```
let x = (...n) => {
    console.log(n);
}

x(4,6,7); // [4, 6, 7]
```



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;
});
```

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

- can be written as

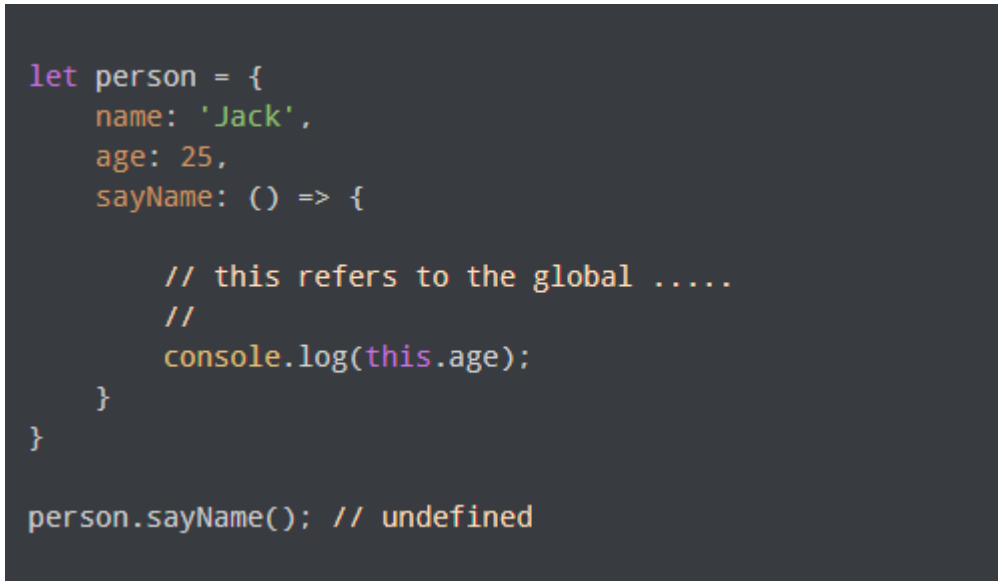
```
// ES6
asyncFunction()
.then(() => asyncFunction1())
.then(() => asyncFunction2())
.then(() => finish);
```


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

- Things You Should Avoid With Arrow Functions

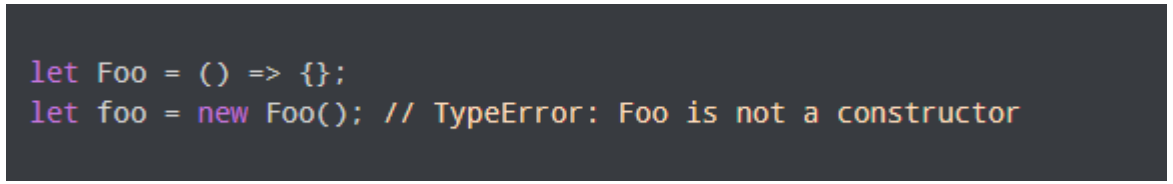
1. You should not use arrow functions to create methods inside objects.

```
let person = {
    name: 'Jack',
    age: 25,
    sayName: () => {

        // this refers to the global .....
        //
        console.log(this.age);
    }
}

person.sayName(); // undefined
```

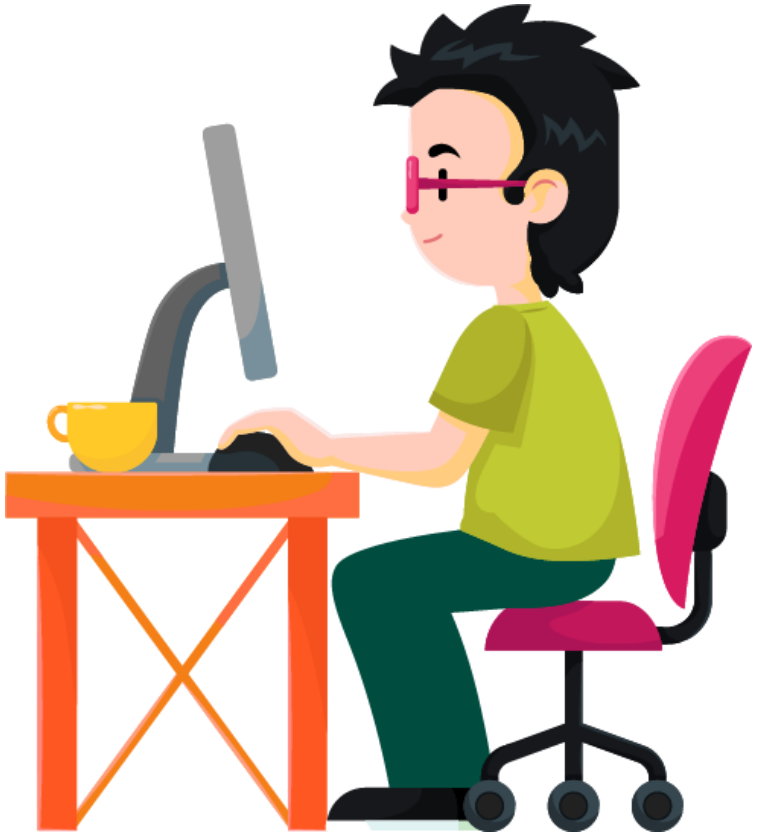


2. You cannot use an arrow function as a constructor. For example,

```
let Foo = () => {};
let foo = new Foo(); // TypeError: Foo is not a constructor
```



Get Exclusive Video Tutorials



www.aptutorials.com
https://www.youtube.com/user/Akashtips

Get More Details

www.akashsir.com

If You Liked It !

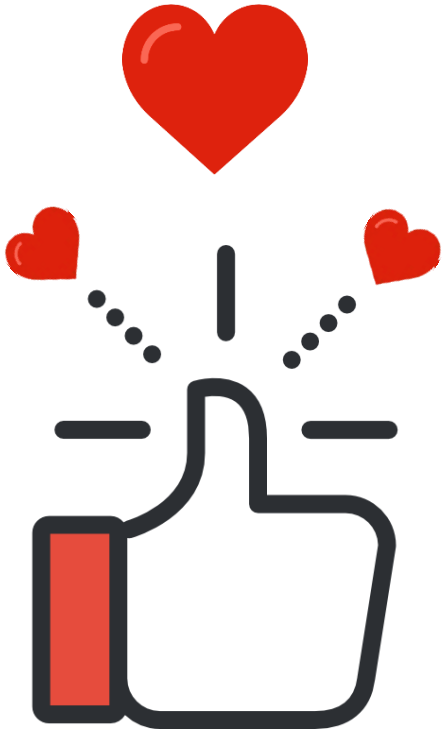
Rating Us Now

Just Dial

https://www.justdial.com/Ahmedabad/Akash-Technolabs-Navrangpura-Bus-Stop-Navrangpura/079PXX79-XX79-170615221520-S5C4_BZDET

Sulekha

https://www.sulekha.com/akash-technolabs-navrangpura-ahmedabad-contact-address/ahmedabad



Connect With Me

Akash Padhiyar
#AkashSir

www.akashsir.com
www.akashtechnolabs.com
www.akashpadhiyar.com
www.aptutorials.com

Social Info

Akash.padhiyar

Akashpadhiyar

Akash_padhiyar

+91 99786-21654

#Akashpadhiyar
#aptutorials