## Variable declaration: var, const, let

In JavaScript, there are three keywords available to declare a variable, and each has its differences. Those are `var`, `let` and `const`.

### Short explanation

Variables declared with `const` keyword can't be reassigned, while `let` and `var` can.
I recommend always declaring your variables with `const` by default, and with `let` if you need to *mutate* it or reassign it later.

| | Scope | Reassignable | Mutable | Temporal Dead Zone |
|---|---|---|---|---|
| **const** | Block | No | Yes | Yes |
| **let** | Block | Yes | Yes | Yes |
| **var** | Function | Yes | Yes | No |

### Sample code

```
const person = "Nick";
person = "John" // Will raise an error, person can't be reassigned

let person = "Nick";
person = "John";
console.log(person) // "John", reassignment is allowed with let
```

### Detailed explanation

The *scope* of a variable roughly means "where is this variable available in the code".

#### var

`var` declared variables are *function scoped*, meaning that when a variable is created in a function, everything in that function can access that variable. Besides, a *function scoped* variable created in a function can't be accessed outside this function.

I recommend you to picture it as if an *X scoped* variable meant that this variable was a property of X.

```
function myFunction() {
  var myVar = "Nick";
  console.log(myVar); // "Nick" - myVar is accessible inside the function
}
console.log(myVar); // Throws a ReferenceError, myVar is not accessible outside the
function.
```

Still focusing on the variable scope, here is a more subtle example:

```
function myFunction() {
  var myVar = "Nick";
  if (true) {
    var myVar = "John";
    console.log(myVar); // "John"
    // actually, myVar being function scoped, we just erased the previous myVar value
"Nick" for "John"
  }
  console.log(myVar); // "John" - see how the instructions in the if block affected
this value
}
console.log(myVar); // Throws a ReferenceError, myVar is not accessible outside the
function.
```

Besides, *var* declared variables are moved to the top of the scope at execution. This is what we call var hoisting.

This portion of code:

```
console.log(myVar) // undefined -- no error raised
var myVar = 2;
```

is understood at execution like:

```
var myVar;
console.log(myVar) // undefined -- no error raised
myVar = 2;
```

**let**

`var` and `let` are about the same, but `let` declared variables

- are *block scoped*
- are **not** accessible before they are assigned
- can't be re-declared in the same scope

Let's see the impact of block-scoping taking our previous example:

```
function myFunction() {
  let myVar = "Nick";
  if (true) {
    let myVar = "John";
    console.log(myVar); // "John"
    // actually, myVar being block scoped, we just created a new variable myVar.
    // this variable is not accessible outside this block and totally independent
    // from the first myVar created !
  }
  console.log(myVar); // "Nick", see how the instructions in the if block DID NOT
affect this value
}
console.log(myVar); // Throws a ReferenceError, myVar is not accessible outside the
function.
```

Now, what it means for *let* (and *const*) variables for not being accessible before being assigned:

```
console.log(myVar) // raises a ReferenceError !
let myVar = 2;
```

By contrast with *var* variables, if you try to read or write on a *let* or *const* variable before they are assigned an error will be raised. This phenomenon is often called *Temporal dead zone* or *TDZ*.

**Note:** Technically, *let* and *const* variables declarations are being hoisted too, but not their assignation. Since they're made so that they can't be used before assignation, it intuitively feels like there is no hoisting, but there is. Find out more on this very detailed explanation here if you want to know more.

In addition, you can't re-declare a *let* variable:

```
let myVar = 2;
let myVar = 3; // Raises a SyntaxError
```

**const**

`const` declared variables behave like *let* variables, but also they can't be reassigned. To sum it up, *const* variables:

- are *block scoped*
- are not accessible before being assigned
- can't be re-declared in the same scope
- can't be reassigned

```
const myVar = "Nick";
myVar = "John" // raises an error, reassignment is not allowed
```

```
const myVar = "Nick";
const myVar = "John" // raises an error, re-declaration is not allowed
```

But there is a subtlety : `const` variables are not **immutable** ! Concretely, it means that *object* and *array* `const` declared variables **can** be mutated.
For objects:

```
const person = {
  name: 'Nick'
};
person.name = 'John' // this will work ! person variable is not completely reassigned, but mutated
console.log(person.name) // "John"
person = "Sandra" // raises an error, because reassignment is not allowed with const declared variables
```

For arrays:

```
const person = [];
person.push('John'); // this will work ! person variable is not completely reassigned, but mutated
console.log(person[0]) // "John"
person = ["Nick"] // raises an error, because reassignment is not allowed with const declared variables
```

#### External resource

- How let and const are scoped in JavaScript - WesBos
- Temporal Dead Zone (TDZ) Demystified

## Arrow function

The ES6 JavaScript update has introduced *arrow functions*, which is another way to declare and use functions. Here are the benefits they bring:

- More concise
- *this* is picked up from surroundings
- implicit return

#### Sample code

- Concision and implicit return

```
function double(x) { return x * 2; } // Traditional way
console.log(double(2)) // 4
```

```
const double = x => x * 2; // Same function written as an arrow function with
implicit return
console.log(double(2)) // 4
```

- *this* reference

In an arrow function, *this* is equal to the *this* value of the enclosing execution context. Basically, with arrow functions, you don't have to do the "that = this" trick before calling a function inside a function anymore.

```
function myFunc() {
  this.myVar = 0;
  setTimeout(() => {
    this.myVar++;
    console.log(this.myVar) // 1
  }, 0);
}
```

### Detailed explanation

#### Concision

Arrow functions are more concise than traditional functions in many ways. Let's review all the possible cases:

- Implicit VS Explicit return

An **explicit return** is a function where the *return* keyword is used in its body.

```
  function double(x) {
    return x * 2; // this function explicitly returns x * 2, *return* keyword is used
  }
```

In the traditional way of writing functions, the return was always explicit. But with arrow functions, you can do *implicit return*which means that you don't need to use the keyword *return* to return a value.

```
  const double = (x) => {
    return x * 2; // Explicit return here
  }
```

Since this function only returns something (no instructions before the *return* keyword) we can do an implicit return.

```
  const double = (x) => x * 2; // Correct, returns x*2
```

To do so, we only need to **remove the brackets** and the **return** keyword. That's why it's called an *implicit* return, the *return*keyword is not there, but this function will indeed return `x * 2`.
**Note:** If your function does not return a value (with *side effects*), it doesn't do an explicit nor an implicit return.

Besides, if you want to implicitly return an *object* you **must have parentheses around it** since it will conflict with the block braces:

```
const getPerson = () => ({ name: "Nick", age: 24 })
console.log(getPerson()) // { name: "Nick", age: 24 } -- object implicitly returned by arrow function
```

- Only one argument

If your function only takes one parameter, you can omit the parentheses around it. If we take back the above *double* code:

```
  const double = (x) => x * 2; // this arrow function only takes one parameter
```

Parentheses around the parameter can be avoided:

```
  const double = x => x * 2; // this arrow function only takes one parameter
```

- No arguments

When there is no argument provided to an arrow function, you need to provide parentheses, or it won't be valid syntax.

```
  () => { // parentheses are provided, everything is fine
    const x = 2;
    return x;
  }

  => { // No parentheses, this won't work!
    const x = 2;
    return x;
  }
```

***this* reference**

To understand this subtlety introduced with arrow functions, you must know how this behaves in JavaScript.

In an arrow function, *this* is equal to the *this* value of the enclosing execution context. What it means is that an arrow function doesn't create a new *this*, it grabs it from its surrounding instead.

Without arrow function, if you wanted to access a variable from *this* in a function inside a function, you had to use the *that = this* or *self = this* trick.

For instance, using setTimeout function inside myFunc:

```
function myFunc() {
  this.myVar = 0;
  var that = this; // that = this trick
  setTimeout(
    function() { // A new *this* is created in this function scope
      that.myVar++;
      console.log(that.myVar) // 1

      console.log(this.myVar) // undefined -- see function declaration above
    },
    0
  );
}
```

But with arrow function, *this* is taken from its surrounding:

```
function myFunc() {
  this.myVar = 0;
  setTimeout(
    () => { // this taken from surrounding, meaning myFunc here
      this.myVar++;
      console.log(this.myVar) // 1
    },
    0
  );
}
```

**Useful resources**

- Arrow functions introduction - WesBos
- JavaScript arrow function - MDN
- Arrow function and lexical *this*

## Function default parameter value

Starting from ES2015 JavaScript update, you can set default value to your function parameters using the following syntax:

```
function myFunc(x = 10) {
  return x;
}
console.log(myFunc()) // 10 -- no value is provided so x default value 10 is assigned to x in myFunc
console.log(myFunc(5)) // 5 -- a value is provided so x is equal to 5 in myFunc
```

```
console.log(myFunc(undefined)) // 10 -- undefined value is provided so default value is assigned to x
console.log(myFunc(null)) // null -- a value (null) is provided, see below for more details
```

The default parameter is applied in two and only two situations:

- No parameter provided
- *undefined* parameter provided

In other words, if you pass in *null* the default parameter **won't be applied**.

**Note:** Default value assignment can be used with destructured parameters as well (see next notion to see an example)

## Class

JavaScript is a prototype-based language (whereas Java is class-based language, for instance). ES6 has introduced JavaScript classes which are meant to be a syntactic sugar for prototype-based inheritance and **not** a new class-based inheritance model (ref).

The word *class* is indeed error prone if you are familiar with classes in other languages. If you do, avoid assuming how JavaScript classes work on this basis and consider it an entirely different notion.

Since this document is not an attempt to teach you the language from the ground up, I will believe you know what prototypes are and how they behave. If you do not, see the external resources listed below the sample code.

### Samples

Before ES6, prototype syntax:

```
var Person = function(name, age) {
  this.name = name;
  this.age = age;
}
Person.prototype.stringSentence = function() {
  return "Hello, my name is " + this.name + " and I'm " + this.age;
}
```

With ES6 class syntax:

```
class Person {
  constructor(name, age) {
```

```
    this.name = name;
    this.age = age;
  }

  stringSentence() {
    return `Hello, my name is ${this.name} and I am ${this.age}`;
  }
}

const myPerson = new Person("Manu", 23);
console.log(myPerson.age) // 23
console.log(myPerson.stringSentence()) // "Hello, my name is Manu and I'm 23
```

**External resources**

For prototype understanding:

- Understanding Prototypes in JS - Yehuda Katz
- A plain English guide to JS prototypes - Sebastian Porto
- Inheritance and the prototype chain - MDN

For classes understanding:

- ES6 Classes in Depth - Nicolas Bevacqua
- ES6 Features - Classes
- JavaScript Classes - MDN

## `Extends` and `super` keywords

The `extends` keyword is used in class declarations or class expressions to create a class which is a child of another class (Ref: MDN). The subclass inherits all the properties of the superclass and additionally can add new properties or modify the inherited ones. The `super` keyword is used to call functions on an object's parent, including its constructor.

- `super` keyword must be used before the `this` keyword is used in constructor
- Invoking `super()` calls the parent class constructor. If you want to pass some arguments in a class's constructor to its parent's constructor, you call it with `super(arguments)`.
- If the parent class have a method (even static) called x, you can use `super.X()` to call it in a child class.

**Sample Code**

```
class Polygon {
  constructor(height, width) {
    this.name = 'Polygon';
    this.height = height;
    this.width = width;
  }

  getHelloPhrase() {
    return `Hi, I am a ${this.name}`;
  }
}

class Square extends Polygon {
  constructor(length) {
    // Here, it calls the parent class' constructor with lengths
    // provided for the Polygon's width and height
    super(length, length);
    // Note: In derived classes, super() must be called before you
    // can use 'this'. Leaving this out will cause a reference error.
    this.name = 'Square';
    this.length = length;
  }

  getCustomHelloPhrase() {
    const polygonPhrase = super.getHelloPhrase(); // accessing parent method with
super.X() syntax
    return `${polygonPhrase} with a length of ${this.length}`;
  }

  get area() {
    return this.height * this.width;
  }
}

const mySquare = new Square(10);
console.log(mySquare.area) // 100
console.log(mySquare.getHelloPhrase()) // 'Hi, I am a Square' -- Square inherits from
Polygon and has access to its methods
console.log(mySquare.getCustomHelloPhrase()) // 'Hi, I am a Square with a length of
10'
```

**Note :** If we had tried to use `this` before calling `super()` in Square class, a ReferenceError would have been raised:

```
class Square extends Polygon {
  constructor(length) {
    this.height; // ReferenceError, super needs to be called first!

    // Here, it calls the parent class' constructor with lengths
    // provided for the Polygon's width and height
    super(length, length);

    // Note: In derived classes, super() must be called before you
    // can use 'this'. Leaving this out will cause a reference error.
    this.name = 'Square';
```

```
  }
}
```