

# Programming in JavaScript

## Section 3: ES6 and JavaScript

---

### 3.1 Const and Let Variables

Const and Let variables are a new feature that is introduced into ES6. The syntax is exactly the same as the var variable keyword; however, the behaviour are slightly different and will throw an error depending on the variable type used.

A const variable must be initialised when being declared i.e. we must assign it a value otherwise we would get an error.

A const or constant variable can store a value just like a variable. However, when declaring a const variable we must assign it a value (unlike var which does not require it to be initialised when declared).

```
var name;  
name = "John";  
  
const name; SyntaxError: Missing initializer in const declaration
```

```
const name = "John";
```

The var value can change its value anytime. The const variable on the other hand does not allow you to change the value of the constant variable and will result an error.

```
var name = "John";  
name = "Mary";  
  
const name = "John";  
name = "Mary"; TypeError: Assignment to constant variable.
```

Therefore, when declaring a const variable we must assign it a value when it is declared and we cannot re-assign it a new value at any point in the code hence, as the name indicates, it has a constant value.

The let variable is almost the same as var but similar to const variables there is a difference between var and let variables. A var variable has two scopes a global and a function scope. The let variable has three scopes a global scope, function scope and a block scope.

When we declare a variable outside a function the variable is said to be declared in the global scope. This means any child functions of the code can access this global variable.

```
var global = "I am a global variable";  
  
function sayMyName() {  
  console.log(global);  
};  
  
sayMyName();  
console.log(global);  
  
I am a global variable  
I am a global variable
```

A local scope (or a function scope) is a variable that is only available in the function and is not available outside in the global scope i.e. it is only local to the function it was declared in. The below code will throw a `ReferenceError` on `console.log(local)` because the global scope does not have access to the function variable outside the local scope. Therefore, the variable can only be manipulated inside of the function where it was declared.

```
function sayMyName() {  
  var local = "I am a local variable";  
  console.log(local);  
};  
  
sayMyName();  
console.log(local);
```

**ReferenceError: local is not defined**

The `let` variable functions exactly the same as the `var` variable i.e. it can be initialised at a later point in time and has both the global and function scope.

```
let global;  
global = "I am a global variable";  
  
function sayMyName() {  
  let local = "I am a local variable";  
  console.log(global);  
  console.log(local);  
};  
  
sayMyName();  
console.log(global);  
// console.log(local);
```

The `console.log(local)` code will throw a `ReferenceError` as seen above example for `var`. As we can see the `let` and `var` variables work exactly the same.

I am a global variable  
I am a local variable  
I am a global variable

The only difference as mentioned above is that the `let` variable has a third scope which is called the block scope. In JavaScript a block is anything that starts and closes with curly brackets ( `{ }` ). The code contained in the curly braces is part of that block of code.

The `let` variable inside of a block can only be accessed in the block scope and cannot be accessed outside of the block i.e. in the global scope (as seen below this will return a `ReferenceError`).

```
{  
  let block = "I am a block scope variable";  
  console.log(block);  
};  
  
console.log(block);
```

**ReferenceError: block is not defined**

When the block code completes its execution the `let` variable no longer exists which is why it gives the `ReferenceError`. This behaviour does not apply to `var` variables as demonstrated in the below example:

```
{  
  var block = "I am a block scope variable";  
  console.log(block);  
};  
  
console.log(block);
```

I am a block scope variable  
I am a block scope variable

Finally, it is important to note that a `let` and `var` variable cannot have the same name (no variables can be declared with the same name) and this will throw a `SyntaxError` to inform you that the variable name has already been declared. In the below example the variable name is called 'variable' and this will change to whatever the name of the variable is that has already been declared.

**SyntaxError: Identifier 'variable' has already been declared**

## 3.2 Block Scope Functions

When a function is created outside in the global scope and another function with the same name is created in a block scope, the new block scoped function will override the global scoped function.

Remember a block scope is anything that starts and ends with the curly brackets e.g. if statements, while, do while and for each loops.

```
function foo() {
  return "b";
};

{
  function foo() {
    return "bar";
  };
  console.log(foo());
};

console.log(foo());
```

```
{
  function foo() {
    return "bar";
  };
  console.log(foo());
};

function foo() {
  return "b";
};

console.log(foo());
```

In this example the `foo()` function in the block scope has overwritten the `foo()` function in the global scope and will print 'bar' in the console twice.

Note the same does not apply in the vice versa i.e. if we declare a function inside of the block scope first and then in the global scope declare a function with the same name, this will not override the block scope.

```
bar
bar
```

```
{
  function foo() {
    return "foo";
  };

  {
    function foo() {
      return "bar";
    };

    console.log(foo());
  };

  console.log(foo());
};
```

When we have a block inside another block (nested blocks) the same behaviour occurs i.e. whereby the new function with the same name overrides the previous function as seen in the example below:

The `console.log()` in the inner block returns 'bar' to the terminal/console because the inner `foo` function overrides the outer block `foo` function.

However, the `console.log()` in the outer main block returns 'foo' in the terminal/console and is not overwritten by the inner block function.

```
{
  let variable = "foo";
  function foo() {
    return variable;
  };

  {
    let variable = "bar";
    function foo() {
      return variable;
    };

    console.log(foo());
  };

  console.log(foo());
};
```

The reason for this behaviour is because the inner block code i.e. functions will only exist in the inner block execution and will override the outer block function with the same name. Once the inner block execution completes the inner function (code block) no longer exists. Therefore, the function in the outer block is now the only function that exists in the block execution.

Therefore, when working with inner block functions that have functions with the same name as the outer block functions we need to pay special attention to this behaviour of JavaScript. This behaviour also applies to variables within block scope.

```
bar
foo
```

**Important Note:** If the inner block 'variable' did not have the 'let' keyword then this will overwrite the outer blocks let variable and would end up printing 'bar' twice.

### 3.3 Optional (Default Value) Parameters

Before ES6 when a function had parameters it was required to pass in all the function parameters within the round brackets when calling the function.

```
function peopleToString(name, age) {  
  return "Your name is: " + name + " and your are " + age + " years old.";  
};  
  
console.log(peopleToString("John", 35));    Your name is: John and your are 35 years old.
```

Now in ES6 if a parameter is not passed into a function the parameters will be replaced by the undefined data types as seen below:

```
console.log(peopleToString());Your name is: undefined and your are undefined years old.
```

ES6 also allows us to create optional parameters. To do this we simply assign a default value to the optional parameter(s) as demonstrated below. This will now use the default value unless the parameter is passed in which will override the default value.

```
function peopleToString(name, age = 30) {  
  return "Your name is: " + name + " and your are " + age + " years old.";  
};  
  
console.log(peopleToString("John"));    Your name is: John and your are 30 years old.  
console.log(peopleToString("John", 35));    Your name is: John and your are 35 years old.
```

### 3.4 Spread Operator

The spread operator ( ... ) within the function parameters allows the function to receive an unlimited parameters. To have an unlimited parameters we would use the spread operator followed by the name of the parameter. This will act as an array where we can pass in any number of parameters.

```
function people(name, age = 45, ...family) {  
  console.log(family);  
};  
  
people("John Doe", 35, "Julie Doe", "Barry Doe");
```

In this example, every other parameter values passed in after the age parameter will be added into the family array.

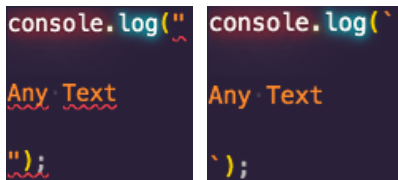
```
[ 'Julie Doe', 'Barry Doe' ]
```

This is a very powerful operator to make function parameters dynamic and have an unlimited amount of parameter that can be passed in based on the scenario. The spread operator in JavaScript is used with arrays. We can loop through the array to do some interesting things with the data.

```
function people(name, age = 45, ...family) {  
  console.log("Your name is " + name + " and you are " + age + " years old. Your family members are:");  
  for(var i = 0; i < family.length; i++) {  
    console.log(family[i]);  
  };  
};  
  
people("John Doe", 35, "Julie Doe", "Barry Doe");  
  
Your name is John Doe and you are 35 years old. Your family members are:  
Julie Doe  
Barry Doe
```

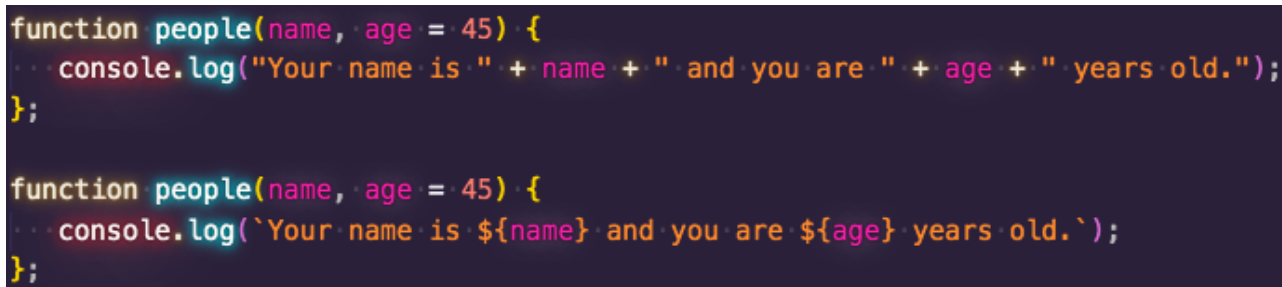
### 3.5 Template Literal

A template literal is a new powerful type of string which has many benefits over the regular string data type. To create a template literal we use double back ticks (``) instead of the single or double quotations.

Two side-by-side screenshots of a code editor. The left screenshot shows a regular string: `console.log("Any Text");`. The right screenshot shows a template literal: `console.log(`Any Text`);`. Both show the text "Any Text" in the console output.

A regular string does not allow you to add empty new lines which will end up as a `SyntaxError`. The template literal on the other hand does not have this problem and will print the new lines.

Strings require the add operator ( + ) to concatenate strings with variables. This syntax can look very unusual or unreadable for humans. Template literal overcomes this by using a special syntax of a dollar sign and curly brackets ( \${ } ) which contains the variable name inside of the curly brackets in order to concatenate to the string. This results in a more easy syntax to write and read as demonstrated in the examples below:

A code editor showing two function definitions. The first function uses string concatenation: `function people(name, age = 45) { console.log("Your name is " + name + " and you are " + age + " years old."); }`. The second function uses template literals: `function people(name, age = 45) { console.log(`Your name is ${name} and you are ${age} years old.`); }`.

### 3.6 Binary Numbers

ES6 introduces the use of binary numbers inside of JavaScript. Binary numbers are part of the numerical binary system. In the real world we use the decimal numeric system which compose numbers with digits between 0 to 9. In the binary number system the numbers are composed of 0s and 1s.

To use a binary number in JavaScript we simply use 0b followed by the binary numbers for example the table below shows the binary number for 1 to 5 as an example:

Binary Number	Decimal Number Equivalent
0b001	1 (i.e. 1.0)
0b010	2 (i.e. 2.0)
0b011	3 (i.e. 3.0)
0b100	4 (i.e. 4.0)
0b101	5 (i.e. 5.0)

Mathematical operators can be used with binary numbers and we can also perform math operations even when mixing both the binary number system with the decimal number system.

```
console.log(0b101 * 0b010); 10  
console.log(0b101 * 3); 15
```

Binary numbers must start with 0b followed by numbers that are between 0 and 1. Any number above 1 or below 0 will result in a SyntaxError.

---

### 3.7 ES6 New JSON Features

```
let name = "John Doe";  
let age = 20;  
  
people = {  
  name: name,  
  age: age  
};
```

In ES5 to assign a variable to a JSON object property's value we required to write the object property name followed by a colon ( : ) followed by the variable name. Where the property name and the variable share the same name this causes a small code repetition.

```
let name = "John Doe";  
let age = 20;  
  
people = {  
  name,  
  age  
};
```

ES6 introduces a shorthand whereby we can emit the variable name as the value if it shares the same name as the object property name. This results in a much more neat and readable code. The object properties takes the values from the variable defined above automatically via the variable name.

```
let animal = {  
  type: "Dog",  
  sound: function() {  
    console.log("bark");  
  }  
};  
  
animal.sound();
```

In ES5 to create a function within an object we had to create a property and then assign the value to a function. To call on the object function we would write the object name followed by the period ( . ) followed by the property name containing the function.

```
let animal = {  
  type: "Dog",  
  sound() {  
    console.log("bark");  
  }  
};  
  
animal.sound();
```

ES6 introduces a new way to work with functions inside of a JSON object which make it more simpler and readable i.e. you no longer need to declare a property name followed by the function( ) keyword. Instead you treat the property name as the function instead.

These are the two new features introduced in ES6 when working with JSON objects i.e. the object property shorthand and object functions syntax.

---

### 3.8 Destructuring

Destructuring is a new feature introduced in ES6 which makes it simpler to take the property values from an object and place them into variables. In ES5 to do this we would have had to use the syntax as seen in the below example:

```
var person = {  
  name: "John Doe",  
  age: 30,  
  eyeColour: "Blue"  
};  
  
var name = person.name;  
var age = person.age;  
var eyeColour = person.eyeColour;  
  
console.log(`Your name is ${name} and you are ${age} years old and have ${eyeColour} eyes.`);
```



The problem with this approach is that the code is much longer and should you decide to change the name of the property in the object then the property name would also need updating for the variable values that reference the object property. This can soon become very cumbersome in a larger code base.

ES6 introduces destructuring which makes this much simpler to do in JavaScript with very little syntax. To destructure an object's properties we start with the variable keyword (i.e. either `var`, `let` or `const`) followed by curly brackets. Inside of the curly brackets the variable names are defined i.e. these are the properties we wish to destructure from the object. Finally, using the assign operator (`=`) we assign the object we wish to destructure the property values from to store in the variables we defined in the curly brackets.

```
var person = {  
  name: "John Doe",  
  age: 30,  
  eyeColour: "Blue"  
};  
  
var { name, age, eyeColour } = person;  
  
console.log(`Your name is ${name} and you are ${age} years old and have ${eyeColour} eyes.`);
```

In the example above, the syntax tells JavaScript to go to the `person` object and extract the `name`, `age` and `eyeColour` property values and assign it to the (`var`) variables which are also named `name`, `age` and `eyeColour`.

**Important Note:** The variables on the left of the assign operator do not have to be in the same order as the properties defined in the object as long as the variable name is the same as the property name this will continue to work.

This is a much cleaner and simpler code for creating new variables and assigning it the values from an object. However, you would still have the same issue as before if you were to change the property name of an object but will be much easier to update the code than using the ES5 syntax.

---

### 3.9 Classes In ES6

ES6 introduces classes which are used for Object Oriented Programming (OOP). A classification is the process that groups properties and actions of a group of objects of animate/inanimate things. Therefore, classes are a way to categorise similar entities into groups. We can think that a class is a mould for an object.

We can define objects' attributes inside a class for example the `People` class can have attributes of `name`, `age`, `height`, `weight`, etc.

We can also define actions that the `people` class can perform for example `walk()`, `jump()`, `run()`, etc. When we have functions in a class this is known as a method.

We can think of a class as an empty table and when we create a new object that uses a class we can think of it as filling the table with the information required for the class. Therefore, we can use classes to create custom data types.

To create a class in ES6 we need to use the class keyword followed by the class name. It is common practice to create a class name where the first character is uppercase. This will help distinguish a variable from a class object. We then use curly brackets to define the properties and methods of the class.

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

Every class needs to have at least one property and to define these we require a special class method called constructor. The constructor is a special method of class within JavaScript.

To define the properties we use the this keyword followed by a period ( . ) and then the name of the property.

The constructor is a special method that gets executed when we create a new object from the class i.e. it is the first method that gets executed when we create a new object from the class. To create a new object from the class we would create a new variable and set its value using the new keyword followed by the class name. The new keyword creates a new copy of the class.

```
let john = new Person();  
john.name = "John Doe";  
john.age = 35;  
console.log(john.name);  
John Doe
```

To access a property from a class object we use the new object name followed by a period ( . ) and then the property name. To set a value to a class property on the new object we use the assign operator to assign the property a value.

We can think of a property as a variable inside of an object. We can create as many new objects from the Person class for each person and each person object will have a name and age property as seen in the above example.

This approach of defining object properties is not efficient and clean. Instead, we can define object properties values in the constructor using parameters. We can then pass in the property values when we instantiate a new object from the class as seen in the example to the right.

```
class Person {  
  constructor(name, age) {  
    this.name = name;  
    this.age = age;  
  };  
};  
  
let john = new Person("John Doe", 35);  
console.log(john.name);
```

To create a method inside of a class is the same as creating a function within the class object i.e. we define a method name followed by a open and closing round brackets followed by curly brackets. To call a class method is the same as calling a property on an object, instead we call on the method as seen in the example to the right. Therefore, a method is simply a function inside of an object.

```
class Person {  
  constructor(name, age) {  
    this.name = name;  
    this.age = age;  
  };  
  
  sayHello() {  
    console.log("Hello");  
  };  
};  
  
let john = new Person("John Doe", 35);  
john.sayHello();
```

```
sayHello(input) {  
  console.log(input);  
};  
  
let john = new Person("John Doe", 35);  
john.sayHello("Hi");
```

We can define parameters inside of a method to make the methods more dynamic. When calling on the method on the object we can pass in arguments which makes the output dynamic.

Methods can also access object properties. The this keyword refers to the object. Therefore, when we call on the this keyword it is telling JavaScript to get the property of the object that called on the sayHello method.

```
sayHello(input) {  
  console.log(`${this.name} says: ${input}`);  
};  
  
let john = new Person("John Doe", 35);  
john.sayHello("Hi");
```



---

## 3.10 Static Methods

The static keyword makes a method in an object a static method. This means that we can access the method by referencing the class itself without having to create a new object of the class.

```
class Person {  
  constructor(name, age) {  
    this.name = name;  
    this.age = age;  
  };  
  
  static onePlusTwo() {  
    console.log(1 + 2);  
  };  
};  
  
Person.onePlusTwo();
```

Without the static keyword this would have meant that we would have had to create a new object of the class and then access the method via the object.

In the example on the left, because the method has the static keyword we are able to call on the method from the class itself without having to create an object.

If we try to call on a method that is not defined as static from the class itself this will cause a Uncaught TypeError.

Therefore, a static method belongs to the class and not the object. If we try to access the static method from an object we would get an Uncaught TypeError error message as demonstrated below.

```
let john = new Person("John Doe", 35);  
john.onePlusTwo();  
TypeError: john.onePlusTwo is not a function
```

---

## 3.11 Getters & Setters

Getters and Setters are special methods (similar to static methods) and allow us to manipulate properties and encapsulate them. Getters are used to get data from the object and Setters are used to define data in properties of the object.

To create a getter we would use the get keyword followed by method name that is the same name as a property existing on the class. A getter must always return some data.

To use a getter we simply would call the object followed by a period ( . ) followed by the property name as a variable. However, this would throw a TypeError because in JavaScript for every getter there must also be a setter.

To create a setter we would use the set keyword followed by the method name that should also be the same name as the property existing on the class and the getter method. A set method requires an argument parameter for the method.

The setter is used to define the data (value) of the property while the getter is used to get the data from the property of the object.

The variable within the code block of the get and set methods cannot use the same property name as the method name because this would throw an Uncaught RangeError.

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

  get age() {
    return this._age;
  };

  set age(age) {
    this._age = age;
  };
};

var john = new Person("John", 35);
console.log(john.age);
```

Every time you try to access a property in the object in the class this will access the setter and this behaviour makes the setter method recursive causing the RangeError.

To fix and good practice to this issue is to use the underscore ( \_ ) before the property name (i.e. you cannot have the same property name inside the getter and setter methods).

The getter and setter will now work without throwing the Uncaught RangeError.

To use a setter we simply would call the object followed by a period ( . ) followed by the property name as a variable. We would then assign a value.

```
var john = new Person("John", 35);
john.age = 20;
console.log(john.age);
```

In the example on the left we set the age property of the john object from 35 to 20 using the setter method.

Therefore, when we call on the property and use the assign operator JavaScript knows to call the setter method to assign a value to the object property while calling on the property without the assign operator will call the get method to return the value of the property. This is how we can use getters and setters with our class objects.

We have more power with getters and setters because we can manipulate the data and encapsulate the attributes/properties. Encapsulation involves providing methods that can be used to read from or write to the field rather than accessing the field directly. This allows to manipulation of the input and output data as well as validate input to maintain data integrity.

```
set age(age) {
  if(age > 0 && age < 100) {
    this._age = age;
  } else {
    this.age = 10;
  };
};
```

In this example on the left the setter method would validate the data passed to ensures that the age property cannot be set to a number above 100 ensuring data integrity of the object.

This is the main power and purpose behind getters and setters.