What you will learn in Module 4

We continue exploring JavaScript Object Oriented Programming:

- You will learn new things about JavaScript object properties and methods
- You will see how to build many instances of objects using ES5 constructor functions and the new ES6 classes
- You will learn how to use the "new" keyword for building objects, and about the mysterious "this" keyword you already met in some examples
- You will learn about "class properties and methods" (vs. "instance properties and methods")
- Finally, you will use these new concepts for improving the game we started to develop in Module 2!

Objects (part 2): properties and methods,

Introduction

You're already familiar with the concept of objects, but so far we've only seen one simple form, called "objects literals" or "singleton objects". I think we've referred to them as "simple objects" in the course. Here is an example:

```
var js1 = {
   courseName: 'JavaScript intro',
   weeks: 5,
   madeBy: 'W3Cx',
   author: 'Michel Buffa' // no "," after the last property!, even if ES5/6 accept it
}
```

And we access properties values using the "." operator, like this:

```
js1.author
js1.weeks
```

Objects (part 2): properties and methods,

Introduction

However, we haven't explained 90% of what is going on, and what we can do with "objects". Our objective in this module, is to explain the most important features of objects, while keeping it simple (more advanced topics will be taught in a future "JavaScript Advanced" course, such as prototypes, context binding, etc.).

Features you will learn:

- The relationship between JavaScript objects and arrays,
- What a "reference" is in a programming language,
- How to embed methods in your objects (functions inside an object),
- The "this" object that you very often encounter in Object Oriented JavaScript code,
- How to add methods and properties to your objects,
- How to make multiple objects of the same class using ES6 classes,
- The built-in JavaScript objects and classes: Array, String, RegExp, Date, Math, Error, etc. And, I will remind you about objects such as navigator, document, window, screen, etc.

Objects (part 2): properties and methods,

From objects to arrays

In Javascript, an object = a table whose keys/indexes are defined!

Look at this **array**:

```
var darkVador = ['villain', 'half human half machine'];
darkVador[0]
darkVador[1]
```

And now, look at this **object**:

```
var darkVador = {
   job: 'villain',
   race: 'half human half machine'
};
```

They look a bit similar, don't they?

Objects (part 2): properties and methods,

From objects to arrays

They look a bit similar, don't they?

- Same name of the variable that contains the object = darkVador
- Instead of '**[**' and '**]**' that we used for defining an array, we use '**{**' and '**}**' for defining an object
- The elements of the object (its properties) are separated by a comma '**,**'
- The pairs of keys/values are separated by '**:**' as in race: 'half human, half machine'
- The last pair of keys/values has no ',' at the end.

It is possible to access the object's properties with "." or with brackets

We saw that we can use the "." operator, followed by the property name. It's also possible to use the bracket notation, and manipulate the object as an array whose indexes, instead of being 0, 1, 2 etc., are the property names!

Objects (part 2): properties and methods,

From objects to arrays

```
var book = {
   title: 'Le Petit Prince',
   author: 'Saint-Exupery'
   };
book['title'] == book.title // you will get true
```

Note: Trick to remember literal object

In JavaScript, objects are arrays whose indexes are property names: please remember this!

Objects (part 2): properties and methods,

Property declaration syntax

Property names: different possibilities

We can put single or double quotes around the name of the property, or nothing at all:

```
var louis = {age: 40}; // WE DO THIS MOST OF THE TIME!
var louis = {"age": 40};
var louis = {'age': 40};
```

In some cases we have to put quotes around the property name:

- When it is a reserved word from JavaScript,
- Or it contains spaces or special characters,
- Or it begins with a number.

Objects (part 2): properties and methods,

Property declaration syntax

Examples:

```
book.1stPublication = '6 avril 1943'; // begins with a numbe // Throws a SyntaxError
book['1stPublication'] = '6 avril 1943'; // OK
book.date of publication = '6 avril 1943'; // spaces not allowed!
book['date of publication'] = '6 avril 1943'; // allowed, but avoid!
```

Another classic case where the name of a property is in a variable, In this case it is necessary to use the syntax with '[' and ']' …

Example:

```
var key = 'title';
book[key];
```

Objects (part 2): properties and methods,

An object can contain another object

Example:

```
var book = {
   name: 'Catch-22',
   published: 1961,
   author: {                    // embedded object!
       givenName: 'Joseph',
       familyName: 'Heller'
   }
};
book.author.givenName;  //equivalent to book['author']['givenName']
book.author.familyName;
```

Accessing the embedded object author is done by chaining property accesses using the "." operator, like in book.author.givenName (here we access the givenName property of the object author, which is also a property of the book object).

Objects (part 2): properties and methods,

Elements, properties and methods

Some vocabulary:

- For **arrays**, we speak of **elements**
- For **objects**, we talk about **properties**
- But a **property** can also be a function, in which case it is called a **method**

Yes, it is possible for an object's property to be a function!

A very simple example:

```
var medor = {
    name: 'Benji',
    bark: function(){
        alert('Ouarf, Ouarf!');
    }
};
```

In this example, the bark property's value is a function, so we call bark "a method".

Objects (part 2): properties and methods,

Elements, properties and methods

A method is a special property that corresponds to the object's behavior

- **Properties** correspond to an **object's DNA** (its characteristics), and **are nouns** (age, name, etc.)
- **Methods** correspond to an **object's behavior** and **are verbs** (bark, move, changeSpeed, etc.)

Calling a method, Since a method is a property we can use the '.' operator (or brackets with the method's name as a string index). Check **example.IV.4.2.00.callingMethod**.

Now that we've seen that we can include methods into objects, here is a better, more readable and more encapsulated version of our player object:

Objects (part 2): properties and methods,

Elements, properties and methods

```
var player = {
    x:10,
    y:10,
    width:20,
    height:20,
    color:'red',
    move(x, y) {
    // change x and y coordinates of the player
   // TODO!
    },
    draw() {
    // draw the player at its current position
   // with current width, height and color
   // TODO!
    }
}
```

Assuming that the move and draw methods are fully implemented, we will now be able to call:

- player.move(mousePos.x, mousePos.y) to change the position of the player,
- player.draw() to draw the player at its current position, with its current size and color.

Readability is better, it is like asking the player to move, or asking it to draw itself. And we do not need to pass the x, y, width, height, color to the draw method: it is inside the player object, and it can access all its internal property values!

In the next section we will look at how we can access other object's properties from a method or call other methods.

Objects (part 2): properties and methods,

"this": accessing properties

The **this** keyword: Accessing properties from a method

The **this keyword**! When one wants to access an object property or wants to call another method from an object method, we must use the this keyword. In the code of the player object, this means "from this object".

Let's look at our game again **example.IV.4.2.01.this.Property**, with a new version of the player object - this time fully functional:

Notice that we've used **this** followed by the '**.**' operator every time we've had to access the current value of an object's property (lines 13, 14, 24, 26 and 28).

We passed the canvas' graphic context as a parameter to the draw method (it's always good not to create dependencies when making objects). Passing the context as a parameter avoids using it as a global variable. If in another project we've got a context named "context" instead of "ctx", then we will just change the parameter when we call player.draw, otherwise we would have had to rename all occurrences of ctx in the code). Same with the mouse coordinates we passed to the move method.

Objects (part 2): properties and methods,

Adding/deleting properties and methods

Properties and methods can be added/deleted **after an object has been defined.** Unlike other object-oriented languages, it is possible in JavaScript to add or to remove properties after an object has been created

Example:

```
// empty object with properties/methods
var darkVador = {};
// add properties after darkVador has been created
darkVador.race = 'human';
darkVador.job = 'villain';
// add some methods
darkVador.talk = function() {
  return 'come to the dark side, Luke!' + this.breathe();
};
```

Objects (part 2): properties and methods,

Adding/deleting properties and methods

Deleting a property or a method: You can use the JavaScript keyword "**delete**" to delete an object's property (it will become undefined).

Example:

```
function deleteSomeProperties() {
   delete darkVador.race;
   delete darkVador.job;
   }
```

Objects (part 3): creating multiple objects, Classes: definition

Introduction: the concept of "class" in object oriented programming languages: So far in this course, we've only used **singleton objects**: objects that **only occur once**: player, darkVador, etc.

Ok, this is not quite true, I'd forgotten that we created many balls in the module 2 game. We'll come back to this example further down the page!

But even with the balls from module 2, we did not use a template to tell us how to easily create multiple objects that share the same properties and the same methods, but whose properties' values may differ.

For example, imagine Luke Skywalker, Ian Solo and Dark Vador. What do they have in common? They all are **Star Wars heroes**, they all have a name, they all belong to one side (the good/bad people, or rebels vs empire), etc. Imagine that we have a way of programming that describes not the objects themselves, but a "model", a "template" for these objects. We could call it **StarWarsHero** and use it for creating our heroes' objects.

Objects (part 3): creating multiple objects, Classes: definition

Imagine the balls from module 2: they all had the same shape (circle), the same x, y, radius and color properties, but they were all different. They all belonged to the same class of object (ball), but they were all different in terms of their properties' values.

In many programming languages, these templates are called "classes".

- In JavaScript 5 (also called ES5), we did not have such a concept, instead we had "constructor functions".
- In JavaScript 6 (ES6), we have the concept of classes, and the syntax is rather similar to what we find in other object oriented programming languages.

Objects (part 3): creating multiple objects, ES5's **constructor functions**, the "new" keyword

ES5's constructor functions, the new keyword

With JavaScript version 5 (and previous versions), you can define a pseudo-class template called "a constructor function". The syntax is the same as for creating a function, except that:

- **By convention, its name is Capitalized.** The first letter of the function name is in uppercase, this is a good way to know, when you read someone else's code, that this is not a regular function, but a constructor function. **Its name is a noun, the name of the class of objects you are going to build.** Example: Person, Vehicle, Enemy, Product, Circle, Ball, Player, Hero, etc.
- **You build new objects using the new keyword**

  Examples (Car, Hero, Ball, Product are constructor function names):

```
var car = new Car('Ferrari', 'red');
var luke = new Hero('Luke Skywalker', 'rebels");
var ball1 = new Ball(10, 10, 20, 'blue'); // x=10, y=10, radius = 20, color = 'blue'
var p1 = new Product('Epson printer P1232', '183', 'Mr Buffa'); // ref, price, customer
```

Objects (part 3): creating multiple objects, ES5's **constructor functions**, the "new" keyword

- **The parameters of the function are the "constructor parameters": the new object that you are building will take these as its initial properties' values.** You can build a Hero, but you must give him/her a name, a side, etc.
- **You define the property names and method names using the this keyword**. But beware: the syntax is not the same as the syntax we used for singleton/simple objects. No more "**:**" and "**,**" between properties. Here we use "**=**" and "**;**" like in regular functions. In a constructor function named "Hero", you will find properties declared like this: this.name this.side; and methods declared like this: this.speak = function() {...}

  Example:

```
function Hero(name, side) {
    this.name = name;
    this.side = side;
    this.speak = function() {
        console.log("My name is " + this.name + " and I'm with the " + this.side);
    }
}
```

Objects (part 3): creating multiple objects, ES5's **constructor functions**, the "new" keyword

- **Very often some properties are initialized using the constructor function parameters,** so that the newly constructed objects will get an initial value for their properties. In this case, we use the this keyword to distinguish the property from the constructor function parameter:

  Example:

```
function Hero(name) {
    this.name = name;
    ...
}
```

Objects (part 3): creating multiple objects, ES5's **constructor functions**, the "new" keyword

```
function Hero(name, side) {
   this.name = name; //code outside of methods is usually
for initializing
   this.side = side;//the properties. Very often, they
match the parameters
   this.speak = function() {
   return "<p>My name is " + this.name +
          ", I'm with the " + this.side + ".</p>";
   }
}
```

Look how the constructor function is declared: the function name starts with an uppercase letter 'Hero'. The parameters have the same name as the properties they correspond to (name, side). And in the first source code lines after the function declaration, we initialize some properties using these parameters (lines 2 and 3). We use the this keyword to distinguish the property and the parameter. You will often see things like: this.name = name; this.age = age; etc.

```
var darkVador = new Hero("Dark Vador", "empire");
var luke = new Hero("Luke Skywalker", "rebels");
var ianSolo = new Hero("Ian Solo", "rebels");
function makeHeroesSpeak() {
  document.body.innerHTML += darkVador.speak();
  document.body.innerHTML += luke.speak();
  document.body.innerHTML += ianSolo.speak();
}
```

Creation of three heroes. We use the same constructor function (Hero) along with the **new** keyword. Luke, darkVador and ianSolo ARE each a Hero, and share the same properties (name, side) and the same behavior (they can speak, they all have a speak method).

Objects (part 3): creating multiple objects, Creating objects using the new **ES6 classes**

ES5's constructor function syntax is not easy to read. If someone does not respect the "conventions" that we've just discussed (start the class with an uppercase, etc.), then the code may work, but it will be difficult to guess that we are not in front of a regular function.

ES6 created a **class keyword** and a **constructor keyword**. Main changes:

- A class is simply defined using the **keyword class** followed by the name of the class
- The unique constructor is defined using the **constructor keyword** followed by the parameters
  - The constructor is executed when an object is created using the keyword new
  - Example: let h1 = new Hero('Ian Solo', 'rebels');
  - This will call constructor(name, side) in the example below.
- A method is simply defined by its name followed by its parameters (**we don't anymore use the keyword "function"**); Example: speak() {...} in the source code below.

Objects (part 3): creating multiple objects, Creating objects using the new **ES6 classes**

```
class Hero {
  constructor(name, side) {
      this.name = name; // property
      this.side = side; // property
  }
  speak() { // method, no more "function"
      return "<p>My name is " + this.name +
          ", I'm with the " + this.side + ".</p>";
  }
}
var darkVador = new Hero("Dark Vador", "empire");
```

Objects (part 3): creating multiple objects, You must declare a class before using it!

Unlike functions, classes must be declared BEFORE using them. An important difference between function declarations and class declarations is that function declarations are "hoisted" and class declarations are not. This means that **you can call a function BEFORE it has been declared in your source code**. This is not the case with ES6 classes!

You first need to declare your class and then access it, otherwise code like the following will throw a **ReferenceError**:

Incorrect version => you try to create an instance of a class before it has been declared:

```
var p = new Rectangle(); // ReferenceError
class Rectangle {...}
```

Correct version =>

```
class Rectangle {...}
var p = new Rectangle(); // It works
```

Objects (part 3): creating multiple objects, Creating objects with functions (**factories**)

We have already seen three different ways to create objects (**literals**, **constructor functions** and **ES6 classes**)

Objects can be created as "literals" :

```
var darkVador = { firstName:'Dark', lastName:'Vador'};
```

Objects can be created with the keyword new and a constructor function or an ES6 class:

```
var darkVador = new Hero('Dark Vador', 'empire');
```

Here is a new one: **objects can also be created by functions that return objects (factories):**

Objects (part 3): creating multiple objects, Creating objects with functions (**factories**)

```
function getMousePos(event, canvas) {
    var rect = canvas.getBoundingClientRect();
    var mxx = event.clientX - rect.left;
    var my = event.clientY - rect.top;

    return { // the getMousePos function returns an object. It's a factory
        x: mx,
        y: my
    }
}
```

And here is how you can use this:

```
var mousePos = getMousePos(evt, canvas);
console.log("Mouse position x = " + mousePos.x + " y = " + mousePos.y);
```

Objects (part 3): creating multiple objects, Static properties and methods

Class properties and methods vs. instances' properties and methods, Sometimes, there are methods "attached" to a class, not to an instance of a class.

For example, imagine the Hero class we've already seen, and we would like to know how many Star Wars's heroes have been created. If zero hero has been created, it's obvious that we could not use this property with an instance of the class such as Dark Vador: darkVador.getNbHeroes(); this would make no sense.

Instead, object oriented programming languages have the concept of "**class properties**" and "**class methods**" that complete the "**instance properties**" and "**instance methods**" that we've seen up to this point. Hero.getNbHeroes() means "Hey, class Hero, can you tell me how many heroes have been created using your class?". Class methods define the "class behavior", and instance methods define the instances' behavior. darVador.speak(); means "Hey, Dark Vador, please, tell us something!". I speak to Dark Vador and I'm expecting something creative from him, such as "I'm your father, Luke!".

Objects (part 3): creating multiple objects, **Static** properties and methods

It's the same for properties. If there is a property named nbHerosCreated in the class Hero, it represents the DNA of the class, not of the instances. You can say "the Hero class has the number of heroes it created", and you can say "Dark Vador has a name and belongs to the empire side", but not "Dark Vador has a number of heroes he created". We have class properties and instance properties.

The **static keyword** is used for defining **class methods**

Class methods; How do we distinguish them? By using the static keyword. When you see a method preceded by the static keyword, it means that you see a class property or a class method.

- The static keyword defines a static method for a class.
- Static methods are called without instantiating their class
- and can not be called through a class instance.
- Consequence: do not use instance properties in their body!
- Static methods are often used to create utility functions for an application (source: MDN).

Objects (part 3): creating multiple objects, **Static** properties and methods

Class properties should be defined after the class definition, and declared using the name of the class followed by the **. operator** and the name of the property. Example: **Point.nbPointsCreated** in the example below. A best practice is to ALWAYS use them this way.

There is another way to declare Class properties (using **static getters** and **setters**).

Objects (part 3): creating multiple objects, Static properties and methods

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
        // static property
        Point.nbPointsCreated++;
    }

    // static method
    static distance(a, b) {
        const dx = a.x - b.x;
        const dy = a.y - b.y;

        return Math.sqrt(dx*dx + dy*dy);
    }
}
```

```
// static property definition is necessarily
outside of the class with ES6
Point.nbPointsCreated=0;

// We create 3 points
const p1 = new Point(5, 5);
const p2 = new Point(10, 10);
const p3 = new Point(12, 27);
document.body.innerHTML = '';
document.body.innerHTML += "<p>Distance between
points (5, 5) and (10, 10) is " +
Point.distance(p1, p2) + "</p>";
document.body.innerHTML += "Number of Points
created is " + Point.nbPointsCreated;
```

Objects (part 3): creating multiple objects, ES6 getters and setters

It is possible to use special methods that are called getters and setters. They allow to make some checks when one is trying to set a value to a property, or to do some processing when accessing it (for example for displaying it in uppercase, even if its value is in lowercase).

These special functions are called "**getters**" and "**setters**", and are declared using the **keywords get** and **set** followed by the name of the property they define.

Objects (part 3): creating multiple objects, ES6 getters and setters

```
class Person {
    constructor(givenName, familyName) {
        this.givenName = givenName; // "normal name"
        this._familyName = familyName; // starts with "_"
    }
    get familyName() {
        return this._familyName.toUpperCase();
    }
    set familyName(newName) {
        // validation could be checked here such as
        // only allowing non numerical values
        this._familyName = newName;
    }
    walk() {
        return (this.givenName + ' ' + this._familyName +
' is walking.'); } }
```

```
let p1 = new Person('Michel', 'Buffa');
console.log(p1.familyName);
// will display BUFFA in the devtool console
// this will call implicitly get familyName();
p1.familyName = 'Smith';
// this will call implicitly set familyName('Smith');
```

Objects (part 3): creating multiple objects, projects

Suggested topics:

- Did you now that ES6 classes are just "a syntactic sugar"? In fact they are equivalent to constructor functions from ES5...
- There are two sorts of object-oriented languages: class-based languages and prototype-based languages.
- JavaScript is a prototype-based language. In this introductory course, we managed to avoid this term! Without getting into too much details, you might be curious about prototypes and maybe read some Web pages related to those.
- And yes, ES6 classes are not "real classes"... They are meant to make developers' lives easier, i.e., for the developers who already know a class-based language such as Java, C#, etc.
- There is a powerful way to define "pseudo classes" using constructor functions: it's called "The Black Box Model". I recommend it to those of you who are rather comfortable with Object Oriented Programming concepts. Give it a try!

Objects (part 3): creating multiple objects, projects

Projects:

- Try to write one of the example from the previous modules without using any single time the keyword "function", use only ES6 classes and instances. In case of problems -> go the the forum and share your experience, this will be very useful for all students to see what sort of problems can occur when moving from a functional approach to an object-oriented approach
- Build an ES6 class-based contact manager
    - Try to build a small database (in a JavaScript array) that will hold your contacts. You will use ES6 classes for defining:
        - a Contact class, with givenName, familyName, phoneNumber, etc. and
        - an HTML set of input fields (not inside a form) for creating new contacts + an "Add contact" button. When you click on the button, it calls an addContact() callback of your own that will create a new contact and add it to your database (using the push method on arrays).

Objects (part 3): creating multiple objects, projects

- input fields and buttons inside a form!

  Beware: either do not put your input fields and buttons inside a <form> or the buttons will submit the form (this is their default behavior, unless you add an attribute type="button" to the buttons). Or you might also declare <form onsubmit = "return processMyForm();">, this will call the method processMyForm (You can change this name if you like) when the form is submitted. In the processMyForm method, get the content of the input fields, build a contact, add it to the array etc. And then, do not forget to return false to avoid the submission of the HTML form).

- It would be cool to also have a listContact() function that will generate a list of contacts (create <ul>... with <li>... inside, one for each contact).
- Now, try to write an ES6 class ContactManager (or you could also use an object literal for that..., but let's try practicing ES6 classes!), that will have the array of contact as a property.

Objects (part 3): creating multiple objects, projects

- Create an instance db of this class: let db = new ContactManager();
- Add in the ContactManager class an add(c) and a list() method (for adding a contact c to the array of contacts, and for listing the contacts).
- Now, when you press the buttons, the addContact() method from step 1 will call db.addContact(c) where db is the instance of your ContactManager class and c is an instance of the Contact class.
- Feel free to customize this project with nice CSS, etc.

Improving the game with classes, ES6 class and constructor

Building balls in our ES6 game: ES6 class and constructor. First, let's look how we were handling balls previously in our game! Building balls in order to fill the array of balls.

OLD VERSION: here is how we built the array of balls

Improving the game with classes, ES6 class and constructor

```
function createBalls(n) {
    let ballArray = []; // empty array
    // create n balls
    for(let i=0; i < n; i++) { // let's build multiple times a singleton object
        let b = {
            x:w/2,
            y:h/2,
            radius: 5 + 30 * Math.random(), // between 5 and 35
            speedX: -5 + 10 * Math.random(), // between -5 and + 5
            speedY: -5 + 10 * Math.random(), // between -5 and + 5
            color:getARandomColor(),
        }
        // add ball b to the array
        ballArray.push(b);
    } // end of for loop
    return ballArray; // returns the array full of randomly created balls
}
```

Improving the game with classes, ES6 class and constructor

In the previous code, in order to build n balls, we created a singleton ball object multiple times. This worked, but if we have misspelled a property name within the code, or forgot one of the properties that had to be initialized, we would have received no warnings. We will replace these lines with something like let b = new Ball(...);

NEW VERSION: using the new keyword and an ES6 class

Improving the game with classes, ES6 class and constructor

```
function createBalls2(n) {
    let ballArray = [];    // empty array
    // create n balls
    for(let i=0; i < n; i++) {
        // Create some random values...
        let x = w/2;
        let y = h/2;
        let radius = 5 + 30 * Math.random(); // between 5 and 35
        let speedX = -5 + 10 * Math.random(); // between -5 and + 5
        let speedY = -5 + 10 * Math.random(); // between -5 and + 5
        let color = getARandomColor();
        // Create the new ball b
        let b = new Ball(x, y, radius, color, speedX, speedY);
        ballArray.push(b); // add ball b to the array
    }
    return ballArray; // returns the array full of randomly created balls
}
```

Improving the game with classes, ES6 class and constructor

Ok, not a very big change here, except that we are no longer manipulating the property names one by one, and we use the new keyword.

And here is the (so far, incomplete) ES6 class for Ball (continued in the next slides of this course)

```
class Ball {
   constructor(x, y, radius, color, speedX, speedY) {
       this.x = x;              // properties
       this.y = y;
       this.radius = radius;
       this.color = color;
       this.speedX = speedX;
       this.speedY = speedY;
   }
... // code to come for methods
}
```

Improving the game with classes, Adding methods to the ES6 class

Ok, we've seen how to define the Ball class: properties and constructor. Properties are the DNA for balls: they all have an x and y position, a radius, a color, a horizontal and a vertical speed. It is time to add some behaviors: a draw and a move method. Indeed, all balls will be able to draw and move themselves.

Here's how we were drawing a ball in the previous version of the game:

```
function drawFilledCircle(c) {

    ctx.save(); // GOOD practice: save the context, use 2D trasnformations
    ctx.translate(c.x, c.y);     // translate the coordinate system, draw relative to it
    ctx.fillStyle = c.color;
     ctx.beginPath();
    ctx.arc(0, 0, c.radius, 0, 2*Math.PI); // (0, 0) is the top left corner of the monster.
    ctx.fill();
    ctx.restore(); // GOOD practice: restore the context
}
```

Improving the game with classes, Adding methods to the ES6 class

And this how we were drawing and moving all the balls:

```
function drawAllBalls(ballArray) {
   ballArray.forEach(function(b) {
       drawFilledCircle(b);
   });
}
function moveAllBalls(ballArray) {
   balls.forEach(function(b, index) { // iterate on all balls in array
       b.x += (b.speedX * globalSpeedMutiplier); // b is the current ball in the array
       b.y += (b.speedY * globalSpeedMutiplier);
       testCollisionBallWithWalls(b);
       testCollisionWithPlayer(b, index);
});
}
```

Improving the game with classes, Adding methods to the ES6 class

Adding a draw and a move method to the ES6 ball class, Instead of having these behaviors as separate functions that take a ball reference as a parameter, it is always better to put this as a method inside the class. Indeed, each ball can move, can draw itself, and the content of these methods does not bring any external dependencies.

For example, if we decide to put a method named testCollisionWithWalls inside the Ball class, it would be bad, in terms of reusability, for its content to rely on external, global variables, such as the canvas size. You could have passed the canvas as a parameter, but then you create more specialization: you have a Ball class for balls that can move inside a rectangular area that is a canvas. It's better to just pass the width and the height of the zone.

Anyway, if you plan to use your balls in another game, it is recommended that you keep the class as simple as possible. It will be more reusable in other projects.

Improving the game with classes, Adding methods to the ES6 class

New version of the ES6 Ball class with draw and move methods:

```
class Ball {
    constructor(x, y, radius, color, speedX, speedY) {  // see previous section for the code
    }
    draw(ctx) {                    // Nearly the same as the old drawFilledCircle function
        ctx.save();                  // BEST practice: save the context, use 2D transformations
        ctx.translate(this.x, this.y);// translate the coordinate system, draw relative to it
        ctx.fillStyle = this.color;
        ctx.beginPath();
        ctx.arc(0, 0, this.radius, 0, 2*Math.PI);     // (0, 0) is the top left corner of the monster.
        ctx.fill();
        ctx.restore();               // BEST practice: restore the context
    }
    move() {
        this.x += this.speedX; this.y += this.speedY; } }
```

Improving the game with classes, Adding methods to the ES6 class

Notice that we did not take into account the globalSpeedMultiplier we had in the old moveAllBalls function, as this is not something that is individually relevant to each ball: it is more something that affect ALL balls. This should raise an alert into your mind: **use an ES6 class property for that,** try to do it!

In other words, even if zero ball has been created, this globalSpeedMultiplier is set and can be modified using a slider in the graphic user interface. Consequently, it is not a ball property, more a property of the Ball class itself.

This setting could be created using a class property, as seen in a previous section of this course.

Improving the game with classes, Adding methods to the ES6 class

And here is how we can now move and draw ALL balls

```
function drawAllBalls2(ballArray) {
    ballArray.forEach(function(b) {
        b.draw(ctx);
    });
}

function moveAllBalls2(ballArray) {
    // iterate on all balls in array
    balls.forEach(function(b, index) {
        b.move(); // b is the current ball in the array
        testCollisionBallWithWalls(b);
        testCollisionWithPlayer(b, index);
});
}
```

Improving the game with classes, Adding methods to the ES6 class

Check **example.IV.4.4.00.ES6.Game** that includes those improvements.