HOCHSCHULE FURTWANGEN UNIVERSITY | HFU

Web Programming
JavaScript – node.js - react

Lecture 2-1:
JavaScript 2
26.10.

Stefan Noll
Manuel Fehrenbach
Winter Semester 22/23

Classes

In the previous chapter, we explored ES6 in depth. In this chapter, we will be learning about one of the most exciting additions to JavaScript – ES6 classes. At the time of writing, classes have become an indispensable part of coding in JavaScript. In simple words, classes are nothing but "*special functions*." As we will go through this chapter, we will understand how and why classes are a beautiful extension of the functions that we have learned earlier in this book. Learning classes will be of immense help if you intend to learn front-end libraries like ReactJS or VueJS.

Structure

In this chapter, we will cover the following topics:

- Understanding classes
- Classes and methods
- Inheritance
- Getters and setters

Objective

After studying this chapter, you should be able to understand how to use JavaScript classes and use them as template to create objects. You will be able to create a rigid structure to encapsulate data.

Understanding classes

ES6 introduced JavaScript to the world of classes. Outside this book, you will learn that ReactJS (one of the most popular front-end libraries in the world, created by Facebook Incorporation) turned classes into a crucial part of the web development language.

Classes, as I mentioned earlier, are a special kind of function. In *Chapter 5: Objects*, we learned about objects and prototypes to create objects. Let's look at that example again:

```
1. function Person(first, last) {
2.   this.firstName = first;
3.   this.lastName = last;
4. }


5.
6. Person.prototype.getFullName = function() {
7.   console.log(`${this.firstName} $
   {this.lastName}`);
8. }
```

Code 7.1: *Objects and Prototypes*

In the preceding *code example 7.1*, we have used a function to create a template for the Object **persons**. Before ES6, the best way to create an object template was to make use of functions. ES6 introduced classes to create a dignified and structured format to generate object templates.

The example shown in the preceding *code example 7.1* can be rewritten as a class in the following way:

```
1. class Person {
2.   constructor(first, last) {
3.     this.firstName = first;
4.     this.lastName = last;
5.   }
6.
7.   getFullName = () => {
8.     console.log(`${this.firstName} $
   {this.lastName}`);
9.   }
10. }
11. let father = new Person('John', 'Doe');
12. father.getFullName();  //John Doe
```

Code 7.2: *Classes*

```
1. function Person(first, last) {
2.   this.firstName = first;
3.   this.lastName = last;
4. }
5.
6. Person.prototype.getFullName = function() {
7.   console.log(`${this.firstName} $
   {this.lastName}`);
8. }
```

Code 7.1: *Objects and Prototypes*

As shown in the *code example 7.2*, we have a more structured approach for creating object templates in the form of classes. To define a class, we use the keyword class, followed by a name for the class, and a pair of curly braces.

Every class always has a default method called *constructor*. The constructor specifies the properties of the class. Even if the user does not explicitly create the method constructor, it always exists. This constructor method is the first thing to be always called when the class object is initialized.

You can also add methods inside the classes. We will learn in detail about the methods in the next section.

Finally, to use a class, you need to create an instance of the class using the JavaScript keyword new, and a pair of parentheses. You can add any number of arguments in the parentheses. By logic, these arguments should be accessed inside the constructor. As shown in the line number 12 of the *code example 7.2*, using the keyword new will create an instance of the class **`Person`**.

```
1. class Person {
2.   constructor(first, last) {
3.     this.firstName = first;
4.     this.lastName = last;
5.   }
6.
7.   getFullName = () => {
8.     console.log(`${this.firstName} $
   {this.lastName}`);
9.   }
10. }
11. let father = new Person('John', 'Doe');
12. father.getFullName();  //John Doe
```

Code 7.2: *Classes*

When to use functions and when to use classes?

In layman terms, the simple difference between functions and classes is as follows:

- **Functions** are a way to package reusable code.
- **Classes** are a blueprint for a JavaScript object.

Therefore, when we are writing code that will be reused several times in the script, we can package it inside a function. However, when we need to create an object that needs to be generated multiple times, we make use of a class to define the object properties and methods.

Are classes hoisted?

In the previous chapters, we have learned that in JavaScript, variable and function declarations are "*moved to the top*" of the script. This "*moving to the top*" is called as *JavaScript hoisting*.

However, unlike variables and functions, JavaScript classes are not hoisted. As shown in the following screen-shot, JavaScript throws an error on the very first line (denoted by the number 1 after the colon in the error highlighted in red). It means that every time you need to use a class, you need to define it first before using it.

```
> let mother = new Person("Jane", "Dias");
  mother.getFullName();

  class Person {
      constructor(first, last) {
          this.firstName = first;
          this.lastName = last;
      }

      getFullName = () => {
          console.log(`${this.firstName} ${this.lastName}`);
      }
  }

  let father = new Person("John", "Doe");
  father.getFullName();

✖ ▶Uncaught ReferenceError: Person is not defined          VM1114:1
      at <anonymous>:1:14
```

Figure 7.1: *Class declarations are not hoisted*

Classes and methods

In *Chapter 5: Objects*, we have learned that the functions defined inside an object are called as *methods*. Since classes are primarily a blueprint for creating objects, functions created inside a class are also called as *methods*. In the preceding *code example 7.2*, we have already seen two methods – **`constructor`** and **`getFullName`**. If you observe the *code example 7.2* closely, you will notice that we do not need to declare the methods created inside a class.

Inside a JavaScript class, there are the following four types of methods:

- **constructor**

The constructor is a special method of the class. It is present by default, even if the developer chooses not to explicitly create it. If the developer chooses not to add a constructor, JavaScript adds an invisible and empty constructor method. The constructor is called automatically when a class is initialized.

The primary job of the constructor is to create and initialize the object of the class. You can specify the properties of the class inside the constructor.

Let's revise the syntax of the constructor with the help of the *code example 7.3*:

```
1. class Person {
2.   constructor(first, last) {
3.     this.firstName = first;
4.     this.lastName = last;
5.     this.nationality = "French";
6.   }
7. }


8.
9. let father = new Person('John', 'Doe');
```

***Code 7.3**: The constructor method of a class*

As shown in the preceding *code example 7.3*, to create a constructor, you need to use the keyword constructor. Do note, only the exact name constructor can be used for creating a constructor. As shown in the *code example 7.3*, the constructor method can take multiple parameters. These parameters are the ones which are passed when a new instance of the class is created using the **new** keyword. As shown on the line number 9 of the *code example 7.3*, the arguments "**John**" and "**Doe**" are passed as parameters to the constructor method.

```
1.class Person {
2.  constructor(first, last) {
3.    this.firstName = first;
4.    this.lastName = last;
5.    this.nationality = "French";
6.  }
7.}

8.
9.let father = new Person('John', 'Doe');
```

Code 7.3: *The constructor method of a class*

Tip: Use the constructor method when you need to initialize data before any of the code in the class is executed.

- **General methods**

We have already seen an example of a general JavaScript method in the *code example 7.2*. Let's understand the methods of a class with another code example:

```
1. class Person {
2.   constructor(first, last) {
3.     this.firstName = first;
4.     this.lastName = last;
5.   }

6.
7.   getFullName = () => {
8.     console.log(`${this.firstName} $
   {this.lastName}`);
9.   }

10.
11.  addNationality() {
12.    this.nationality = "French";
13.  }
14. }

15.
16. let father = new Person('John', 'Doe');
17. father.getFullName();     //John Doe
18. father.addNationality();
19. console.log(father);     //{age: 24,
   firstName: "John", lastName: "Doe"}
```

Code 7.4: *Traditional JavaScript methods*

As shown in the *code example 7.4*, you can either use the ES6 arrow functions or the traditional JavaScript functions. To invoke (call) a method inside a class, you first need to create an instance of the class, as shown in the line number 16 in the *code example 7.4*. Next, you can call the method name by typing the name of the class instance, and then calling the method name with a pair of parentheses. As with general JavaScript methods, you can pass arguments to these methods by adding values inside the parentheses during the method invocation.

If you have been noticing closely, you don't need to declare a method when you use it inside the class. It is because these methods are a part of the class object **`Person`**.

- **Static methods**

  A static method is a special kind of method. You can use a static method in any of the following cases:

  - **No dependency on class instance**: The code inside the method is not dependent on the instance of the class getting generated. In fact, if the code inside the method will not be using the instance variable at all, we should use it as a static method.
  - **Code sharing**: If all the methods in the instance will be sharing a piece of code, it makes more sense to use it as a static method. For instance, if we have a class Area which calculates the area of any geometrical figure, we will probably require a function to restrict the decimal points of any number to just two points. Such a function will be shared by all the methods inside the class for calculations. This function or rather, this method should be a static method.
  - **No change in method definition**: A static method should be used if the definition of the method or the code written in the method should not be changed or overridden.
  - **Utility classes**: If you are writing a class that is not a part of the actual flow of the code, but it helps the actual flow indirectly, it should possess static methods.

Static methods are defined on the class itself, and not on the instance of the class.

To create static methods, we need to add the JavaScript keyword static in front of the method.

As shown in the following screenshot, we have defined a static method **`toTwoDecimals`**. If we call the static method on the class itself, it will execute. However, if you attempt to execute the **`toTwoDecimals`** method on the class instance rectangle, it results in an error.

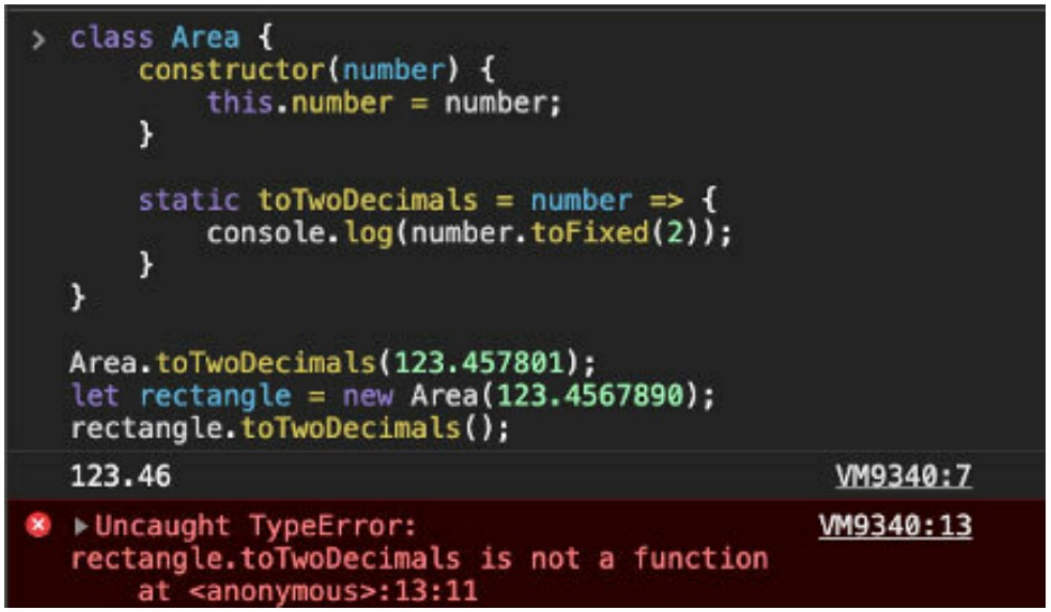

Figure 7.2: *Static methods*

Inheritance

In simple words, inheritance means receiving properties or money from the previous holder. JavaScript follows a similar interpretation. In JavaScript, when you use the JavaScript keyword **extends** during a class definition, it inherits all the methods from the class name that is specified after the keyword **extends**. Along with the keyword, you also need to call the **`super()`** method in the constructor of the child element to access all the properties and the methods of the parent class.

Let's understand inheritance with an example:

```
> class Person {
      constructor(age) {
          this.nationality = "English";
          this.language = "UK English";
          this.age = age;
      }
  }

  class Student extends Person {
      constructor(name, props) {
          super(props);
          console.log(props, name);
          this.name = name;
      }

      display() {
          console.log(`The student's name is ${this.name}. The
  student is a ${this.age} years old ${this.nationality} student and
  speaks fluent ${this.language}.`);
      }
  }

  let student1 = new Student("John Doe", 24);
  student1.display();
  24 "John Doe"                                          VM890:12
  The student's name is John Doe. The student is a 24    VM890:17
  years old English student and speaks fluent UK English.
```

Figure 7.3: *Inheritance*

```
> class Person {
      constructor(age) {
          this.nationality = "English";
          this.language = "UK English";
          this.age = age;
      }
  }

  class Student extends Person {
      constructor(name, props) {
          super(props);
          console.log(props, name);
          this.name = name;
      }

      display() {
          console.log(`The student's name is ${this.name}. The
  student is a ${this.age} years old ${this.nationality} student and
  speaks fluent ${this.language}.`);
      }
  }

  let student1 = new Student("John Doe", 24);
  student1.display();
  24 "John Doe"                                          VM890:12
  The student's name is John Doe. The student is a 24    VM890:17
  years old English student and speaks fluent UK English.
```

Figure 7.3: *Inheritance*

In the preceding screenshot, we have a class **`Person`** having in-built properties, nationality and language set to English and UK English. The **`Person`** class also accepts a parameter age. We also have a class **`Student`** which extends the class **`Person`**. When we use the keyword extends, JavaScript enables a class to inherit properties and methods from another class. In the case of *figure 7.3*, the class **`Student`** calls the super (props) method to access all the properties and the methods of the parent class **`Person`**. Hence, when the method display of the class **`Student`** is called, it displays values from the properties of both the class **`Student`** and the class **`Person`**.

Inheritance is useful for code reusability. It helps reuse of the existing properties and the methods of a class when you create a new class.

Getters and Setters

In the preceding section, we saw four different types of methods used in ES6 Classes. The last category is known as Getters and Setters. Going by the definition of the word itself, getters and setters are the methods inside JavaScript classes to get the value of properties of the class and to set the value of the properties of the class. These methods seamlessly fit into the class definition. However, we can do much more than simply getting and setting the values. These methods can help us in writing some code which needs to be executed before we get or set the value. For instance, if we want to send a request to the server before setting a value, we can do that inside the set method.

To keep it simple, getters and setters is a structured way for fetching and setting the values of the class. We can create a get and a set method using the JavaScript keywords get and set. In the case of the setters, it can also accept parameters to set the property values.

One important thing to keep in mind here is that the name of the getter and the setter method cannot be the same as the name of the property. Let's understand it with an example:

In *figure 7.4*, we have a class **`Person`** with a property name. We have also created a **`get`** method and a **`set`** method. As I had mentioned earlier, you cannot use the same name for the getters and setters as the name of the property. Hence, we have used the name of the method as **`personName`** instead of the property name **`name`**. Kindly do notice that you don't need to explicitly call the get and set method. They work behind the scenes when you try to fetch or set a value of the properties.

Now, let's consider the example of a toaster. At some point in our lives, we must have either used or seen a toaster. It's an easy-to-use product. All you need to do is insert the slices of breads at the designated areas and turn on the machine. When the slice of breads will get toasted, they will pop out. On the exterior, there are two empty spaces to place the bread, a button to pop out the slices of breads, a button to turn on the machine and a knob for setting up the level of toast darkness. However, if I ask you to open up the machine to its complex interiors and then use it for toasting, would you be able to use it?

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

      get personName() {
          console.log("The get method is called.");
          return this.name;
      }

      set personName(name) {
          this.name = name;
      }
  }

  person = new Person("John Doe");
  console.log(person.personName);

  The get method is called.                    VM340:7
  John Doe                                     VM340:17
```

Figure 7.4: *Getters and setters*

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

      get personName() {
          console.log("The get method is called.");
          return this.name;
      }

      set personName(name) {
          this.name = name;
      }
  }

  person = new Person("John Doe");
  console.log(person.personName);

  The get method is called.                    VM340:7
  John Doe                                     VM340:17
```

Figure 7.4: *Getters and setters*

This example shows why it's crucial to keep certain implementations and methods private to a tool, and the strong reason to not expose it to the public. In a similar analogy, we can turn certain variables and methods private to a class. We will look at this in greater detail with the latest introduction of private methods in the second last chapter – ES6 and beyond.

HOCHSCHULE FURTWANGEN UNIVERSITY | HFU

Private class features

Class fields are `public` by default, but private class members can be created by using a hash `#` prefix. The privacy encapsulation of these class features is enforced by JavaScript itself.

Private members are not native to the language before this syntax existed. In prototypical inheritance, its behavior may be emulated with `WeakMap` objects or closures, but they can't compare to the `#` syntax in terms of ergonomics.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Private_class_fields

Syntax


```
class ClassWithPrivateField {
  #privateField;
}

class ClassWithPrivateMethod {
  #privateMethod() {
    return 'hello world';
  }
}

class ClassWithPrivateStaticField {
  static #PRIVATE_STATIC_FIELD;
}

class ClassWithPrivateStaticMethod {
  static #privateStaticMethod() {
    return 'hello world';
  }
}
```

Syntax

```
class ClassWithPrivateField {
  #privateField;
}

class ClassWithPrivateMethod {
  #privateMethod() {
    return 'hello world';
  }
}

class ClassWithPrivateStaticField {
  static #PRIVATE_STATIC_FIELD;
}

class ClassWithPrivateStaticMethod {
  static #privateStaticMethod() {
    return 'hello world';
  }
}
```

Private fields

Private fields include private instance fields and private static fields.

Private instance fields

Private instance fields are declared with **# names** (pronounced "*hash names*"), which are identifiers prefixed with `#`. The `#` is a part of the name itself. Private fields are accessible on the class constructor from inside the class declaration itself. They are used for declaration of field names as well as for accessing a field's value.

It is a syntax error to refer to `#` names from out of scope. It is also a syntax error to refer to private fields that were not declared before they were called, or to attempt to remove declared fields with `delete`.

```
class ClassWithPrivateField {
  #privateField;

  constructor() {
    this.#privateField = 42;
    delete this.#privateField;   // Syntax error
    this.#undeclaredField = 444; // Syntax error
  }
}

const instance = new ClassWithPrivateField()
instance.#privateField === 42;   // Syntax error
```

Note: Use the `in` operator to check for potentially missing private fields (or private methods). This will return `true` if the private field or method exists, and `false` otherwise.

Like public fields, private fields are added at construction time in a base class, or at the point where `super()` is invoked in a subclass.

```
class ClassWithPrivateField {
  #privateField;

  constructor() {
    this.#privateField = 42;
  }
}

class SubClass extends ClassWithPrivateField {
  #subPrivateField;

  constructor() {
    super();
    this.#subPrivateField = 23;
  }
}

new SubClass();
// SubClass {#subPrivateField: 23}
```

Note: `#privateField` from the `ClassWithPrivateField` base class is private to `ClassWithPrivateField` and is not accessible from the derived `Subclass`.

Private instance methods

Private instance methods are methods available on class instances whose access is restricted in the same manner as private instance fields.

```
class ClassWithPrivateMethod {
  #privateMethod() {
    return 'hello world';
  }

  getPrivateMessage() {
    return this.#privateMethod();
  }
}

const instance = new ClassWithPrivateMethod();
console.log(instance.getPrivateMessage());
// hello world
```

Private instance methods may be generator, async, or async generator functions. Private getters and setters are also possible, although *not* in generator, async, or async generator forms.

Conclusion

Classes serve as an excellent way to create object templates. In fact, as I have mentioned earlier, classes turn up to be one of the most significant features of JavaScript when you use front-end libraries like ReactJS or VueJS. In this chapter, we have learned how to create classes and more importantly, where to use classes. We also learned about the different types of methods that we can use in classes. From the next chapter onwards, we will move into a more practical application of advanced JavaScript with reaching out to the server.

Points to remember

- Classes were introduced in ES6 to create a more structured format for writing object templates.
- Classes have four different types of methods.
- The constructor method is the default method inside a class. It is the first thing to be called when an object is initialized. Typically, the constructor is used for setting the initial values of the class.
- Even if the user has not explicitly created the method constructor, it is by default called behind-the-scenes by JavaScript.
- If you have a method, which is not dependent on the class instance or the script inside the method will not be modified, you can use the JavaScript keyword static to turn it into a static method.
- JavaScript classes can inherit the properties and the methods of a class using the **extends** keyword, and calling the `super()` method inside the child's constructor.
- Finally, classes provide an optional separate method for getting or setting the values of the properties of the class. Care must be taken to avoid naming these methods with the name of the properties.