[Lesson 3-4]

Roi Yehoshua 2018

[DAN.IT]
EDUCATION

What we learnt last time?

- Propotypes
- Prototype inheritance
- Native prototypes

[Our targets for today]

- Functions call and apply
- Classes
- Class-based OOP
- Class Inheritance

Call and apply

→ There’s a special built-in function method **func.call()** that allows to call a function explicitly setting **this**

→ The syntax is: `func.call(context, arg1, arg2, ...)`

→ It runs func providing the first argument as this, and the next as the arguments

→ As an example, in the code below we call sayHi in the context of different objects

```
function sayHi() {  alert(this.name);
}

let user = { name: "John" };  let admin = { name:
"Admin" };

// use call to pass different objects as "this"
sayHi.call(user); // this = John  sayHi.call(admin);
// this = Admin
```

Call and apply

→ And here we use call to call say with the given context and phrase:

```
function say(time, phrase) {
    alert(`[${time}] ${this.name}: ${phrase}`);
}

let user = { name: "John" };
say.call(user, '10:00', 'Hello'); // [10:00] John: Hello (this=user)
```

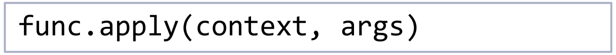
→ There is another built-in method **func.apply()** that works almost the same as func.call(), but takes an array-like object instead of a list of arguments:

```
function say(time, phrase) {
    alert(`[${time}] ${this.name}: ${phrase}`);
}

let user = { name: "John" };
let messageData = ['10:00', 'Hello']; // become time and phrase

// user becomes this, messageData is passed as a list of arguments (time, phrase)
say.apply(user, messageData); // [10:00] John: Hello (this=user)
```

Call and apply

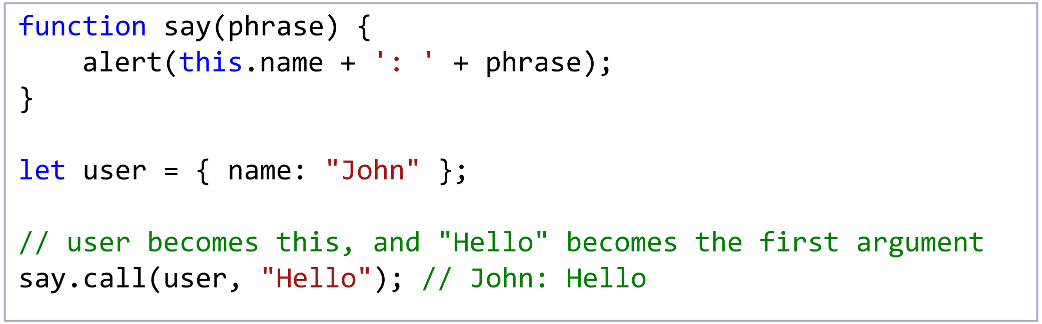
→ There is another built-in method func.apply() that works almost the same as func.call()

→ The syntax is:



```
func.apply(context, args)
```

→ The only syntax difference between call and apply is that call expects a list of arguments, while apply takes an array-like object with them



```
function say(phrase) {
    alert(this.name + ': ' + phrase);
}

let user = { name: "John" };

// user becomes this, and "Hello" becomes the first argument
say.call(user, "Hello"); // John: Hello
```

[Classes in JavaScript]

→ In OOP, a **class** is an extensible program-code-template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods)

→ In JavaScript there are several programming patterns to make classes

→ In ES6, the class construct was introduced, but it’s a “syntax sugar” and an extension of one of the patterns that we’ll study now

[Functional Class Pattern]

→ The constructor function below can be considered a “class” according to the definition:

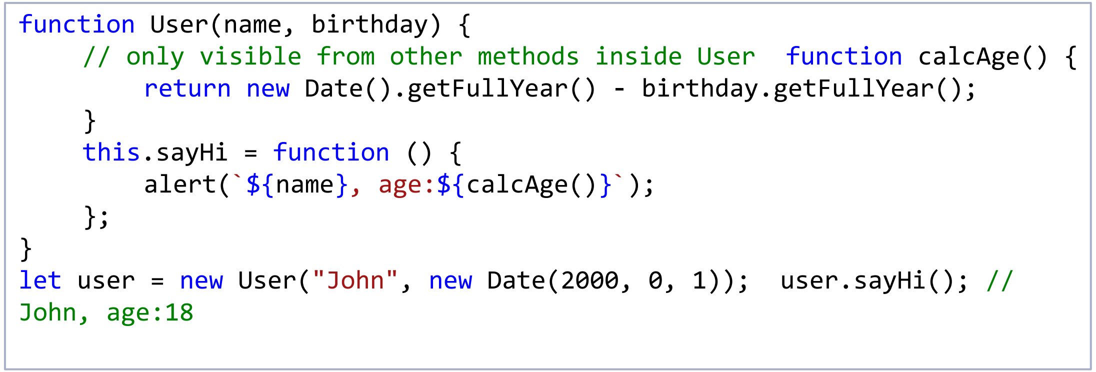
```
function User(name) {  this.sayHi =
    function () {
            alert(name);
    };
}

let user = new User("John");
user.sayHi(); // John
```

→ It follows all parts of the definition:

- → It is a “program-code-template” for creating objects (callable with new)
- → It provides initial values for the state (name from parameters)
- → It provides methods (sayHi)

[Functional Class Pattern]

→ Local variables and nested functions inside User, that are not assigned to **this**, are visible from inside, but not accessible by the outer code



```
function User(name, birthday) {
    // only visible from other methods inside User  function calcAge() {
        return new Date().getFullYear() - birthday.getFullYear();
    }
    this.sayHi = function () {
        alert(`${name}, age:${calcAge()}`);
    };
}
let user = new User("John", new Date(2000, 0, 1));  user.sayHi(); //
John, age:18
```

→name, birthday and the function calcAge() are internal, *private* to the object

- →They are only visible from inside of it

→On the other hand, sayHi is the external, *public* method.

- →The external code that creates user can access it

Factory Class Pattern

→ We can create a class without using new at all

```
function User(name, birthday) {
    // only visible from other methods inside User  function calcAge() {
        return new Date().getFullYear() - birthday.getFullYear();
    }

    return {
        sayHi() {
            alert(`${name}, age:${calcAge()}`);
        }
    };
}

let user = User("John", new Date(2000, 0, 1));  user.sayHi(); // John,
age:18
```

→ The only benefit of this method is that we can omit new: write let user = User(...) instead of let user = new User(...). In other aspects it’s almost the same as the functional pattern.

Prototype-Based Classes

→ Prototype-based classes are the most important and generally the best

```
function User(name, birthday) {  this._name = name;  this._birthday =
    birthday;
}

User.prototype._calcAge = function () {
    return new Date().getFullYear() - this._birthday.getFullYear();
};
User.prototype.sayHi = function () {
    alert(`${this._name}, age:${this._calcAge()}`);
};

let user = new User("John", new Date(2000, 0, 1));  user.sayHi(); //
John, age:18
```

→ The code structure:

- → The constructor User only initializes the current object state
- → Methods are added to User.prototype

[Prototype-Based Classes]

→As we can see, methods are lexically not inside function User

  →If we declare variables inside function User, then they won’t be visible to methods

→ So, there is a widely known agreement that internal properties and methods are prepended with an underscore "_", like _name or _calcAge()

  →Technically, that’s just an agreement, the outer code still can access them

→The advantages over the functional pattern:

  →In the functional pattern, each object has its own copy of every method

    →We assign a separate copy of this.sayHi = function() {...} and other methods in the constructor.

  →In the prototypal pattern, all methods are in User.prototype that is shared between all objects

    →An object itself only stores the data

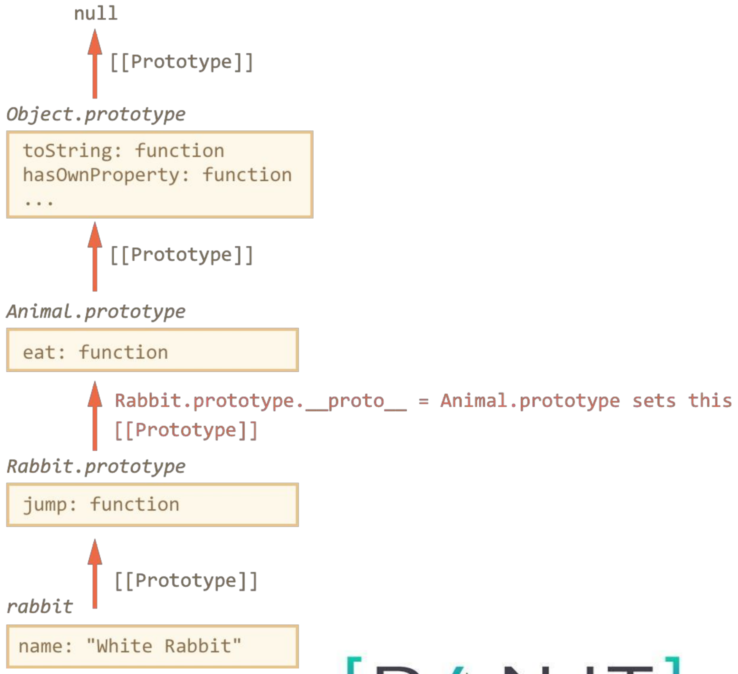
  →Prototypes also allows us to setup the inheritance in a really efficient way (see next slide)

  →Built-in JavaScript objects all use prototypes

[Prototype-Based Inheritance]

→We can set up a prototype inheritance chain between classes and their sub-classes:

```
function Animal(name) {
      this.name = name;
}
Animal.prototype.eat = function () {  alert(`${this.name}
    eats.`);
};

function Rabbit(name) {
      this.name = name;
}
Rabbit.prototype.jump = function () {  alert(`${this.name}
    jumps!`);
};

// setup the inheritance chain
Rabbit.prototype.__proto___= Animal.prototype;

let rabbit = new Rabbit("White Rabbit");  rabbit.eat(); //
rabbits can eat too  rabbit.jump();
```



Classes

→ The “class” construct allows to define prototype-based classes with a clean, nice- looking syntax

→ Here’s a class User and its equivalent prototype-based syntax:

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

    sayHi() {
        alert(this.name);
    }
}

let user = new User("John");
user.sayHi();
```

=

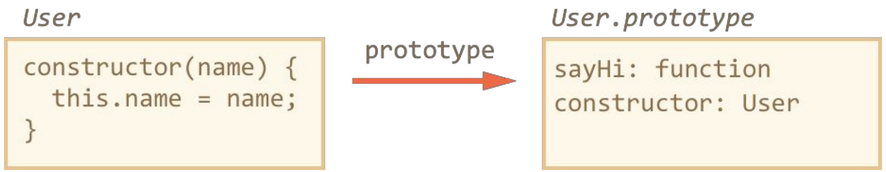
```
function User(name) {  this.name = name;
}

User.prototype.sayHi = function () {
    alert(this.name);
}

let user = new User("John");
user.sayHi();
```

→ Note that methods in a class do not have a comma between them

Classes

→The class User {...} here actually does two things:

- →Declares a variable User that references the function named "constructor"
- →Puts the methods listed in the definition into User.prototype



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

```
sayHi: function
constructor: User
```

→ There are a few subtle differences between the new class syntax and the previous one:

- → Unlike a regular function, a class constructor can’t be called without new
- → Class methods are non-enumerable (they don’t appear in a for..in loop over the objects)
- → A default constructor() {} is generated, if there is no constructor defined in the class construct
- → All code inside the class construct is automatically in strict mode

[Property Getters/Setters]

→ Getters and setters are functions that work on getting and setting a value, but look like regular properties to an external code

→They can be used as wrappers over “real” property values to gain more control over them

→ The getter works when obj.propName is read, the setter – when it is assigned

```
class User {
     constructor(name) {
          this.name = name; // invokes the setter
     }
     get name() {
          return this._name;
     }
     set name(value) {
          if (value.length == 0) {  alert("Name
               cannot be empty");  return;
          }
          this._name = value;
     }
}
```

```
let user = new User("John");  alert(user.name);
// John

user = new User(""); // Name cannot be empty
```

[Methods Only]

→ Unlike object literals, no property:value assignments are allowed inside class

→ There may be only methods and getters/setters

→ If we really need to put a non-function value into the prototype, then we can alter prototype manually, like this:

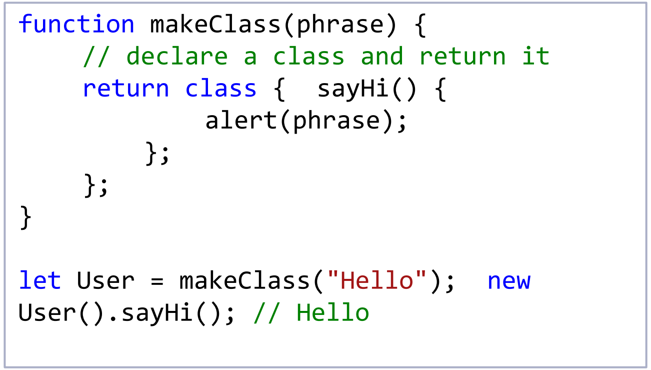
```
class User { }
User.prototype.test = 5;
alert(new User().test); // 5
```

→ Note that such properties will be shared among all objects of the class

Class Expression

→ Just like functions, classes can be defined inside another expression, passed around, returned, etc.

→ For example, here's a class-returning function ("class factory"):



```
function makeClass(phrase) {
    // declare a class and return it
    return class {  sayHi() {
            alert(phrase);
        };
    };
}

let User = makeClass("Hello");  new
User().sayHi(); // Hello
```

→ That's quite normal if we recall that class is just a special form of a function-with-prototype definition

[Static Methods]

→ We can also assign methods to the class function, not to its "prototype“

→ Such methods are called *static*

```
class User {
    static staticMethod() {  alert(this
        === User);
    }
}
User.staticMethod(); // true
```

=

```
function User() { }

User.staticMethod = function () {
          alert(this === User);
};
User.staticMethod(); // true
```

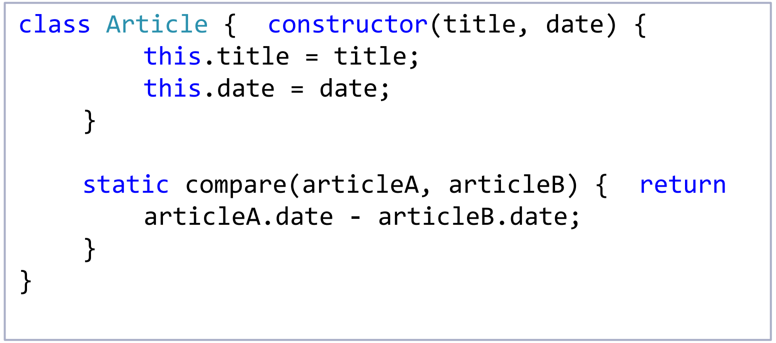
→The value of this inside User.staticMethod() is the class constructor User itself (the “object before dot” rule)

Static Methods

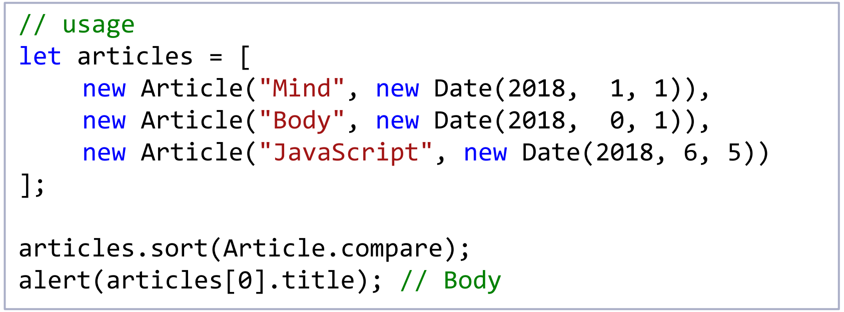
→ Static methods are usually used to implement functions that belong to the class, but not to any particular object of it

→ For instance, we have Article objects and need a function to compare them

→ The natural choice would be Article.compare, like this:



```
class Article {  constructor(title, date) {
        this.title = title;
        this.date = date;
    }

    static compare(articleA, articleB) {  return
        articleA.date - articleB.date;
    }
}
```



```
// usage
let articles = [
    new Article("Mind", new Date(2018,  1, 1)),
    new Article("Body", new Date(2018,  0, 1)),
    new Article("JavaScript", new Date(2018, 6, 5))
];

articles.sort(Article.compare);
alert(articles[0].title); // Body
```

[Summary]

→ The basic class syntax looks like this:

```
class MyClass {
    constructor(...) {
        // ...
    }
    method1(...)  { }
    method2(...)  { }
    get something() { }
    set something(...) { }  static
    staticMethod(..) { }
    // ...
}
```

→ The value of MyClass is a function provided as constructor

→ If there’s no constructor, then an empty function

→ Methods listed in the class declaration become members of its prototype

→ With the exception of static methods that are written into the function itself and callable as MyClass.staticMethod()

[Exercise (1)]

- → Write a class Product with the following properties:
  - →id (a read-only property)
  - →name
  - →price – must be a positive number
- → Add the following methods to the class:
  - →makeDiscount(discount) - changes the price of the product according to the specified discount
  - →print() – prints the product’s details to the console
- → Add a static method to the class that compares two products according to their price
- → Your class code should be in a file product.js
- → In an HTML page create an array of 3 products and sort them by their price
- → Print the products in the array after the sort

Class Inheritance

→ To inherit from another class, you should specify "**extends**" and the parent class before the brackets {..}:

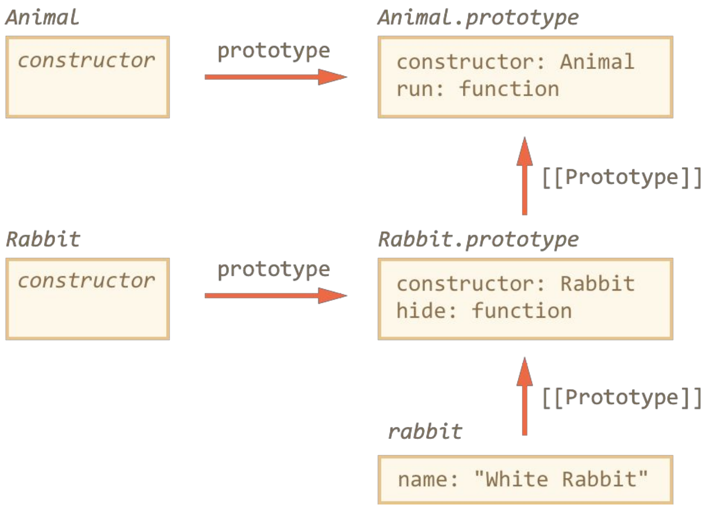
```
// Base class
class Animal {  constructor(name) {
        this.name = name;  this.speed = 0;
    }

    run(speed) {
        this.speed += speed;  alert(`${this.name} runs
        with speed
${this.speed}.`);
    }

    stop() {
        this.speed = 0;  alert(`${this.name}
        stopped.`);
    }
}
```

```
// Inherit from Animal
class Rabbit extends Animal {  hide() {
        alert(`${this.name} hides!`);
    }
}

let rabbit = new Rabbit("White Rabbit");

rabbit.run(5); // White Rabbit runs with speed 5.
rabbit.hide(); // White Rabbit hides!
```

Class Inheritance

→ The **extends** keyword actually adds a [[Prototype]] reference from a Rabbit.prototype to Animal.prototype, as we’ve seen before

→ So now rabbit has access both to its own methods and to methods of Animal



[Overriding a Method]

→ As of now, Rabbit inherits the stop method that sets this.speed = 0 from Animal

→ If we specify our own stop in Rabbit, then it will be used instead:

```
class Rabbit extends Animal {  stop() {
        // ...this will be used for rabbit.stop()
    }
}
```

→ But usually we don’t want to totally replace a parent method, but rather to build on top of it, tweak or extend its functionality

→We do something in our method, but call the parent method before/after it or in the process

→ Classes provide "**super**" keyword for that:

→super.method(...) to call a parent method

→super(...) to call a parent constructor (inside our constructor only)

[Overriding a Method]

→For instance, let our rabbit autohide when stopped:

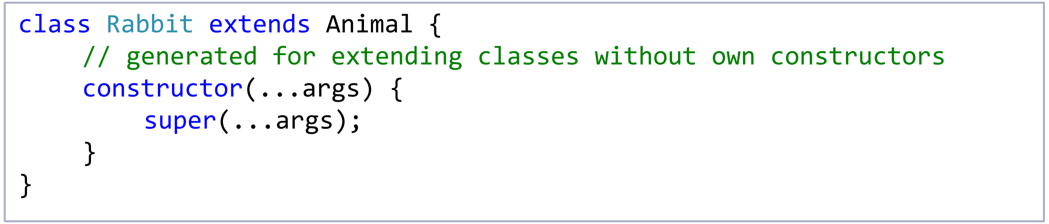
```
class Rabbit extends Animal {
    hide() {
        alert(`${this.name} hides!`);
    }

    stop() {
        super.stop(); // call parent stop
        this.hide(); // and then hide
    }
}
let rabbit = new Rabbit("White Rabbit");  rabbit.run(5); // White


Rabbit runs with speed 5.


rabbit.stop(); // White Rabbit stopped. White rabbit hides!
```

→Now Rabbit has the stop method that calls the parent super.stop() in the process

Overriding Constructor

→ Till now, Rabbit did not have its own constructor

→ If a class extends another class and has no constructor, then the following constructor is generated:



```
class Rabbit extends Animal {
    // generated for extending classes without own constructors
    constructor(...args) {
        super(...args);
    }
}
```

→ As we can see, it basically calls the parent constructor passing it all the arguments

→ Custom constructors must also call super(...), and do it before using this

[Overriding Constructor]

→ For example, let’s add a custom constructor to Rabbit, that will specify the earLength in addition to name:

```
class Rabbit extends Animal {  constructor(name, earLength) {
        super(name);
        this.earLength = earLength;
    }
    // ...
}

let rabbit = new Rabbit("White Rabbit", 10);  alert(rabbit.name); //
White Rabbit  alert(rabbit.earLength); // 10
```

→ For the constructor to work, we need to call super() before using this

Static Methods and Inheritance

→ The class syntax supports inheritance for static properties too

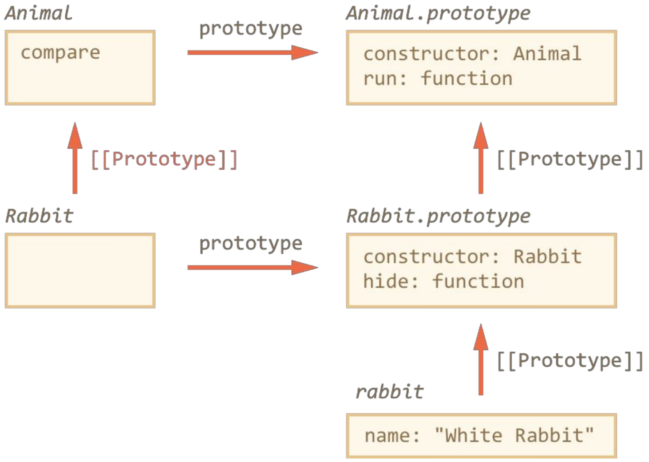
```
class Animal {
      constructor(name, speed) {
            this.speed = speed;
            this.name = name;
      }
      run(speed = 0) {
            this.speed += speed;
            alert(`${this.name} runs with speed
${this.speed}.`);
      }

      static compare(animalA, animalB) {
            return animalA.speed - animalB.speed;
      }
}

// Inherit from Animal
class Rabbit extends Animal {
      hide(){
            alert(`${this.name} hides!`);
      }
}
```

```
let rabbits = [
    new Rabbit("White Rabbit", 10),  new Rabbit("Black
    Rabbit", 5)
];

// We can call Rabbit.compare assuming that the
inherited Animal.compare will be called
rabbits.sort(Rabbit.compare);
rabbits[0].run(); // Black Rabbit runs with speed 5.
```

Static Methods and Inheritance

→ How does it work? Again, using prototypes

→ The keyword **extends** also gives the Rabbit function a [[Prototype]] reference to Animal



→ So, the Rabbit constructor function now inherits from the Animal constructor function

→ which itself has [[Prototype]] referencing Function.Prototype

[Natives are Extendable]

→ Built-in classes like Array, Map and others are extendable too

→ For instance, here PowerArray inherits from the native Array:

```
// add one more method to it (can do more)  class
PowerArray extends Array {
    isEmpty() {
        return this.length === 0;
    }
}

let arr = new PowerArray(1, 2, 5, 10, 50);
alert(arr.isEmpty()); // false

let filteredArr = arr.filter(item => item >= 10);
alert(filteredArr); // 10, 50
alert(filteredArr.isEmpty()); // false
```

→ Note that built-in methods like filter, map, etc. return new objects of exactly the inherited type

→ And we can keep using its methods further down the chain

[Class Checking: instanceof]

→ The instanceof operator allows to check whether an object belongs to a certain class

→ The syntax is:

```
obj instanceof Class
```

→ It returns true if obj belongs to the Class (or a class inheriting from it)

→ For instance:

```
class Animal { }
class Rabbit extends Animal { }

let rabbit = new Rabbit();
alert(rabbit instanceof  Rabbit);  // true
alert(rabbit instanceof  Animal);  // true
alert(rabbit instanceof Object); // true, because Animal inherits from Object
```

→ The instanceof operator examines the prototype chain for the check

[Exercise (2)]

→Create a class Book that extends the Product class from the previous exercise, and adds the following properties to it:

→authors – an array of author names

→pubDate – publication date

→ Override the print() method so after calling the Product's print method, it should print the authors names and the publication date

→ Place your code in the file book.js

→ Test your code in an HTML page that creates an array of 3 books and prints them to the console

[Control questions]

1. What is class?
2. How are classes implemented in JavaScript?
3. What is the purpose of getters and setters?
4. What is static method?
5. How does class inheritance work?
6. How can we extend a class?