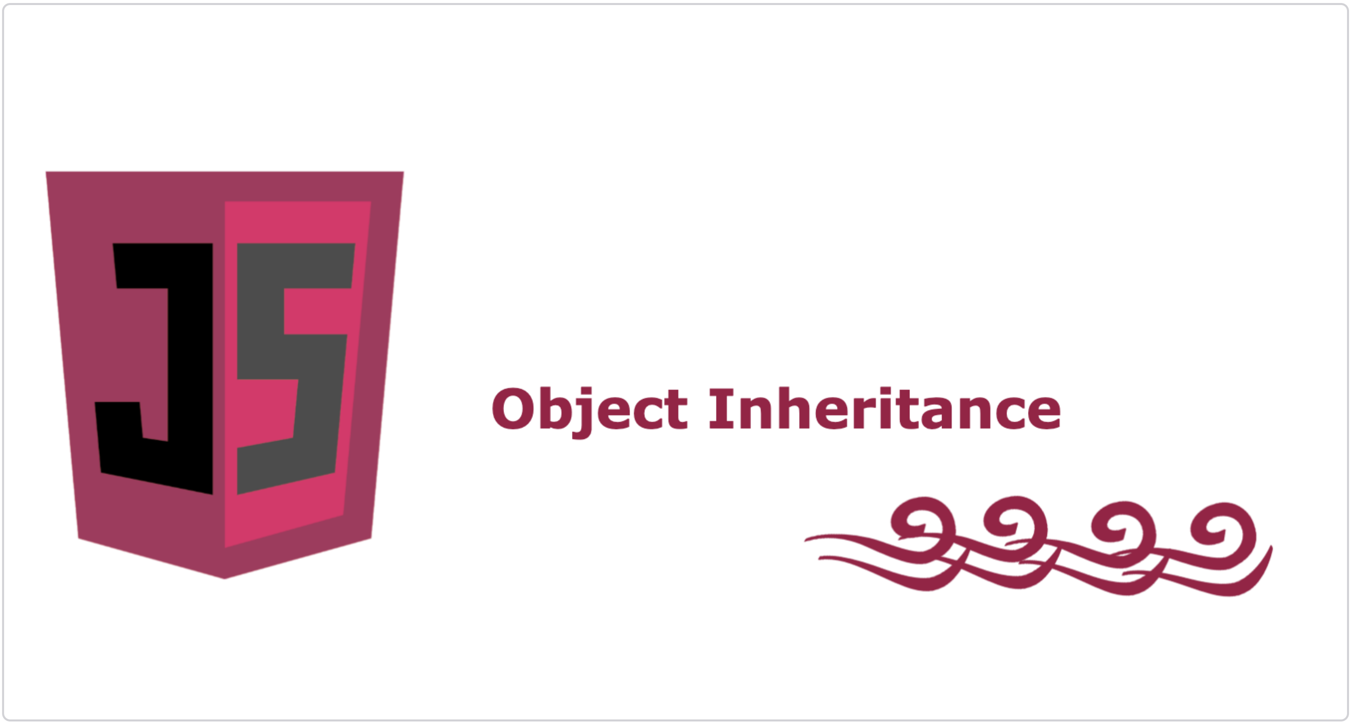
Object Inheritance

In this lesson, we learn how object inheritance works in JavaScript.
Let's begin! :)

WE'LL COVER THE FOLLOWING

- Listing 8-19: Prototype chaining
- Listing 8-20: Pseudo-classical inheritance



Object-oriented programming is unimaginable without inheritance. While most languages support several types of inheritance like implementing interfaces, deriving new types from existing ones, overriding methods, etc., JavaScript supports only implementation inheritance, where actual methods are inherited.

Due to the flexibility and dynamism of JavaScript, there are a number of ways you can implement inheritance. In this section, you'll learn about the two most commonly used patterns, prototype chaining and pseudo-classical inheritance.

Prototype chaining uses a simple trick: you can change the default prototype of a function to an object of your own.

Listing 8-19 demonstrates this technique. It defines three object types (via three constructor functions), `Vehicle`, as the base type, and `Car` and `Boat` as derived types. Observe that derived types set their prototypes to an instance of `Vehicle`.

Listing 8-19: Prototype chaining

```
<!DOCTYPE html>
<html>
<head>
  <title>Prototype chaining</title>
  <script>
    // --- Base type
    var Vehicle = function () {
      this.vendor = "Toyota";
    };

    // --- Base type behavior
    Vehicle.prototype.getVendor = function() {
      return this.vendor;
    };
    Vehicle.prototype.getType = function() {
      return "vehicle";
    };

    // --- Derived type
    var Car = function() {
      this.wheel = 4;
    };

    // --- Chain to base type
    Car.prototype = new Vehicle();

    // --- Override behavior
    Car.prototype.getType = function() {
      return "car";
    };

    // --- Derived type
    var Boat = function() {
      this.propeller = 1;
    };

    // --- Chain to base type
    Boat.prototype = new Vehicle();

    // --- Override behavior
    Boat.prototype.getType = function() {
      return "boat";
    };
    // --- Test
    var car = new Car();
    var boat = new Boat();
    console.log("It's a " + car.getType());
    console.log(car.getVendor());
    console.log("wheels: " + car.wheel);
    console.log("It's a " + boat.getType());
    console.log(boat.getVendor());
    console.log("props: " + boat.propeller);
  </script>
</head>
<body>
  Listing 8-19: View the console output
</body>
</html>
```

When you run this code, the output suggests it really does something that we can achieve in other languages with inheritance:

console

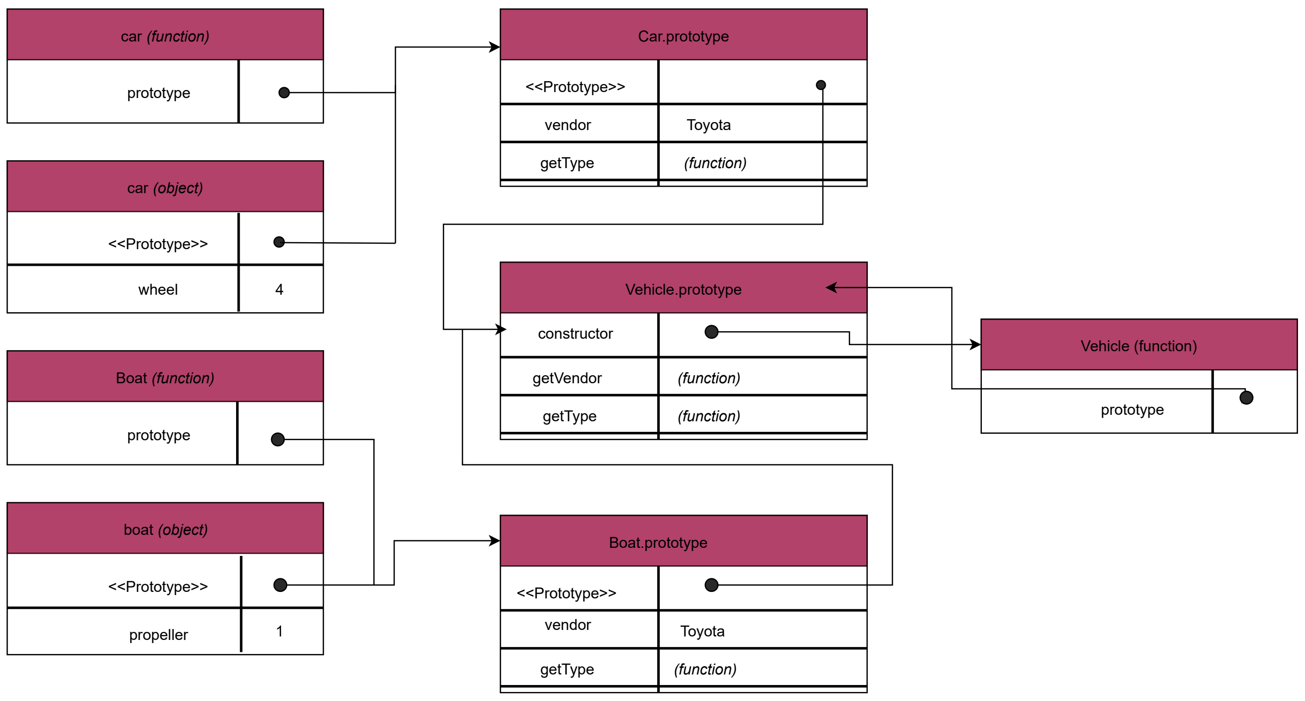
```
It's a car
Toyota
wheels: 4
It's a boat
Toyota
props: 1
```

The trick behind this pattern is the following. `Car` and `Boat` inherit from `Vehicle` by assigning a newly created instance of `Vehicle` and assigning it to `Car.prototype` and `Boat.prototype`. This action overwrites the original prototype of these types.

Having a `Vehicle` instance as the prototype means that all properties and methods that typically exist on an instance of `Vehicle` now also exist on `Car.prototype` and `Boat.prototype`.

As you already know, object properties and methods are resolved in an iterative search that starts with the object instance and provided the property or method has not been found, it goes and searches the prototype. However, the search does not stop here; it traverses to the prototype of the prototype, and along the chain until the property or method is found or the chain ends.

Figure 8-4 depicts the prototype chain, and it helps you understand how inheritance is implemented.



Inheritance via prototype chain

For example, when the boat.propeller or car.wheel properties are passed to `console.log()`, those are found right in the object instances.

When the JavaScript engine looks for the `boat.getType` function, it does not find it within the object instance. The search goes on with the `Boat.prototype`, which happens to be a `Vehicle` instance extended with the `getType` function, where the function is found. When the `car.getVendor` function is looked up, it cannot be found either in the object instance or along the next link in the chain, `Car.prototype`.

Just like `Boat.prototype`, `Car.prototype` is another `Vehicle` instance extended with the Car-specific `getType` function. The search traverses to `Vehicle.prototype`, and there it finds the `getVendor` method.

> 📜**NOTE:** You can extend prototype chaining to create longer inheritance chains. You can use the same technique to create derived types from `Car` and `Boat`.

Figure 8-4 contains a simplification. In reality, `Vehicle.prototype` also contains a `«Prototype»` property that points to `Object.prototype`.

The methods of `Object`, such as `hasOwnProperty()`, `toString()` and others, are defined in that prototype.

Prototype chaining supports the `instanceof` operator.

You can add the following lines to Listing 8-19 to test whether car and boat are instances of the specified types:

```
console.log(car instanceof Car);      // true
console.log(car instanceof Boat);     // false
console.log(car instanceof Vehicle);  // true
console.log(car instanceof Object);   // true

console.log(boat instanceof Car);     // false
console.log(boat instanceof Boat);    // true
console.log(boat instanceof Vehicle); // true
console.log(boat instanceof Object);  // true
```

The `instanceof` operator tests whether an object has the prototype property of the specified constructor in its prototype chain. Because car has the Car, Vehicle, and Object constructor functions on its prototype chain, all related `instanceof` expressions return true.

However, the Boat constructor is not in the prototype chain of car, so car `instanceof` Boat returns false.

Although prototype chaining is simple, it has an annoying issue. You cannot pass arguments to the ancestor types when a derived type is created. In Listing 8-19, the Vehicle constructor does not accept arguments, and it sets the vendor property to a default value, Toyota. This is an ugly solution, just as the implementation of Car and Boat constructors that set the wheel and propeller properties with the same method. Another pattern, pseudo-classical inheritance combines a prototype chain with a technique, constructor stealing, to solve this issue.

Listing 8-20 utilizes this method to improve the code.

Listing 8-20: Pseudo-classical inheritance

```
<!DOCTYPE html>
<html>
<head>
  <title>Pseudoclassical inheritance</title>
  <script>
    // --- Base type
    var Vehicle = function (vendor) {
      this.vendor = vendor;
    };

    // --- Base type behavior
    Vehicle.prototype.getVendor = function() {
      return this.vendor;
    };
    Vehicle.prototype.getType = function() {
      return "vehicle";
    };

    // --- Derived type
    var Car = function(vendor, wheel) {
      Vehicle.call(this, vendor);
      this.wheel = wheel;
    };

    // --- Chain to base type
    Car.prototype = new Vehicle();

    // --- Override behavior
    Car.prototype.getType = function() {
      return "car";
    };

    // --- Derived type
    var Boat = function(vendor, propeller) {
      Vehicle.call(this, vendor);
      this.propeller = propeller;
    };

    // --- Chain to base type
    Boat.prototype = new Vehicle();

    // --- Override behavior
    Boat.prototype.getType = function() {
      return "boat";
    };

    // --- Test
    var car = new Car("BMW", 4);
    var boat = new Boat("Yamaha", 2);

    console.log("It's a " + car.getType());
    console.log(car.getVendor());
    console.log("wheels: " + car.wheel);
    console.log("It's a " + boat.getType());
    console.log(boat.getVendor());
    console.log("props: " + boat.propeller);
  </script>
</head>
```

```
<body>
  Listing 8-20: View the console output


</body>
</html>
```

In this listing, all constructor functions have arguments. `Vehicle` accepts a vendor name while `Car` and `Boat` accept an additional wheel and propeller argument. The `Car` and `Boat` constructors explicitly invoke the `Vehicle` constructor with the `Vehicle.call()` method.

The first argument of `call()` is the object instance to invoke the method on, subsequent arguments represent the parameters of the method to pass. So, the `Vehicle.call(this, vendor)` expression in the `Car` and `Boat` constructors invokes the `Vehicle` function on the newly created object instance with the specified vendor argument.

As a result, the vendor property of the new object instance is set to the passed argument value.

> 📜**NOTE:** There are several other patterns to implement object inheritance in JavaScript. You can easily find them when you search the web for "JavaScript inheritance patterns" with your preferred search engine.

In the *next lesson*, we'll study the concept of modularity.