

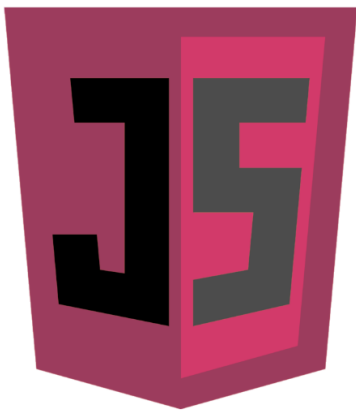
# Using Object Prototypes

In this lesson, we get acquainted with object prototypes and their usage in JavaScript. Let's begin!

## WE'LL COVER THE FOLLOWING



- [Listing 8-16](#): Construction with prototype
- [Listing 8-17](#): Overriding prototype properties
- [Listing 8-18](#): Extending the prototype



## Object Prototypes



Prototypes may provide a great solution for both the resource wasting and poor maintainability issues. As you learned in [Chapter 8](#) (A Quick Overview of Function Prototypes), functions in JavaScript are created with a prototype property, which is an object containing properties and methods that should be available to instances of a particular object type.

Modifying the Employee construction as shown in Listing 8-16 offers a remedy for the issues mentioned earlier.

## Listing 8-16: Construction with prototype #

```
<!DOCTYPE html>
<html>
<head>
  <title>Construction with prototype</title>
  <script>
    var Employee = function (id, firstName,
      lastName, title) {
      this.id = id;
      this.firstName = firstName;
      this.lastName = lastName;
      this.title = title;
    }

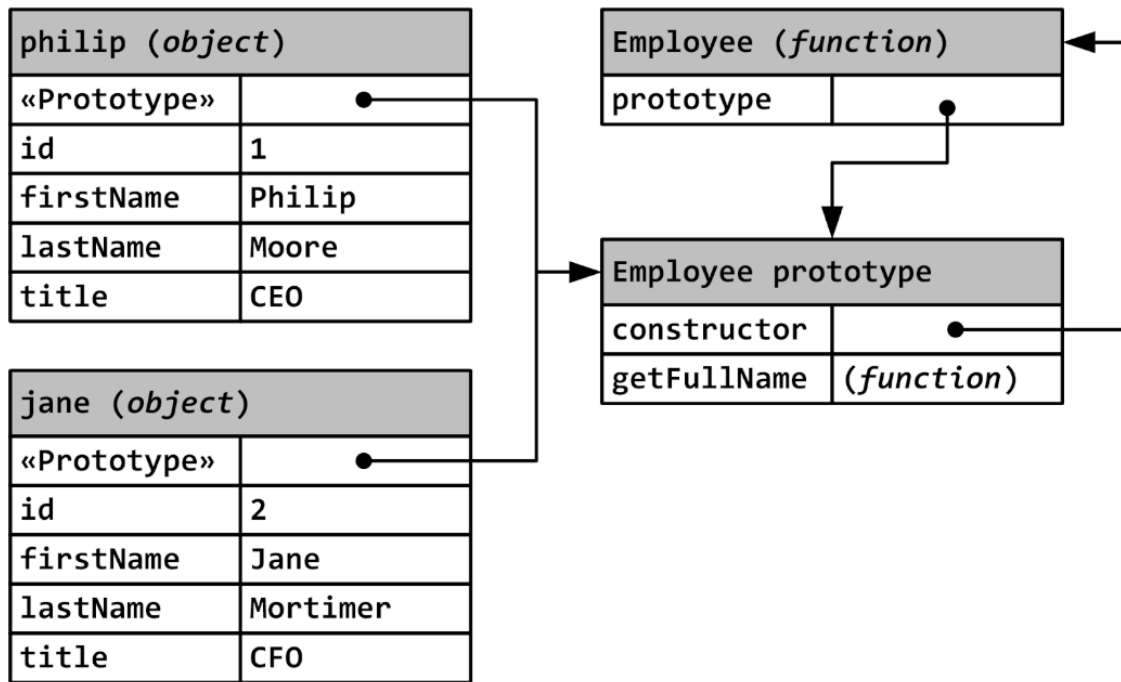
    Employee.prototype.getFullName =
      function () {
        return this.lastName + ", " + this.firstName;
      }

    var philip = new Employee(1, "Philip", "Moore",
      "CEO");
    var jane = new Employee(2, "Jane", "Mortimer",
      "CFO");

    console.log(philip.getFullName());
    console.log(jane.getFullName());
  </script>
</head>
<body>
  Listing 8-16: View the console output
</body>
</html>
```

To understand how prototypes work, here is a short explanation. Each function has a prototype property; whenever a function is created, its prototype property is also set. By default, all prototypes get a property called `constructor` that points to the function used to create the object instance. As shown at the right side of Figure 8-3, the `Employee.prototype` has a `constructor` property that points to the `Employee` function.

Each time the constructor function is called to create a new instance, that instance has an internal pointer to the constructor's prototype (see «Prototype» in Figure 8-3). So, there is a direct link between the instance and the constructor function's prototype.



Employee instances and the prototype

The figure also shows the relationship between the `Employee` constructor's prototype, and the two instances of `Employee`, `philip` and `jane`. Observe that `Employee.prototype` points to the prototype object but `Employee.prototype.constructor` points back to `Employee`. The prototype contains only the constructor property and the `getFullName` method that was added by the highlighted code in Listing 8-16. The `philip` and `jane` instances have a number of internal properties assigned in the constructor function and the internal `«Prototype»` property that points back to `Employee.prototype` only. But they have no direct relationship with the constructor function. When property names are resolved, instances and the constructor's prototype (through the `«Prototype»` property) are looked up.

You can check the link between object instances and prototypes. The `Object.getPrototypeOf()` function retrieves the `«Prototype»` property of the object instance passed as an argument. You can invoke the `isPrototypeOf()` method on a certain constructor's prototype to check whether it is the prototype linked to a specified object instance. You can check these functions by adding these code lines to Listing 8-16:

```
console.log(Employee.prototype
    .isPrototypeOf(philip));
console.log(Employee.prototype
    .isPrototypeOf(jane));
console.log(Object.getPrototypeOf(jane)
    .getFullName === philip.getFullName);
```



All three `console.log()` calls return true. The most interesting is the third one, which proves that the `getFullName` property of philip and jane point to the same function.

You can easily override properties and functions assigned to the prototype, as Listing 8-17 demonstrates.

## Listing 8-17: Overriding prototype properties #

```
<!DOCTYPE html>
<html>
<head>
  <title>Overriding prototype properties</title>
  <script>
    var Employee = function (id, firstName,
        lastName, title) {
      this.id = id;
      this.firstName = firstName;
      this.lastName = lastName;
      this.title = title;
    }

    Employee.prototype.getFullName =
      function () {
        return this.lastName + ", " + this.firstName;
      }

    var philip = new Employee(1, "Philip", "Moore",
        "CEO");
    var jane = new Employee(2, "Jane", "Mortimer",
        "CFO");
    jane.getFullName = function () {
      return "Jane";
    }

    console.log(philip.getFullName());
    console.log(jane.getFullName());
  </script>
</head>
<body>
  Listing 8-17: View the console output
</body>
</html>
```

When the JavaScript engine searches for property names, it first looks up the property in the object instance. If not found, it goes to look up the properties

in the prototype. In Listing 8-17, the `jane` instance defines its own `getFullName`

property, and this property is found before the same property specified in the prototype.

There are situations when you need to know whether a property has been defined on the object, or in the prototype. The `hasOwnProperty()` function of an object instance returns true if the property with the specified name exists on the object instance.

If you'd like a list of all instance properties, you can use the `Object.getOwnPropertyNames()` function. You can examine the application of these methods when you append the following lines to the end of Listing 8-17:

```
console.log(philip
  .hasOwnProperty("getFullName")); // false;
console.log(jane
  .hasOwnProperty("getFullName")); // true;
console.log(Object
  .getOwnPropertyNames(philip).length); // 4
console.log(Object
  .getOwnPropertyNames(jane).length); // 5
```

Because prototypes are objects, you can add properties and methods to the underlying objects at any time. This allows you to extend object types. Listing 8-18 contains a self-explaining code snippet that extends the `Number` and `String` types with two new functions.


## Listing 8-18: Extending the prototype #

```
<!DOCTYPE html>
<html>
<head>
  <title>Extending the prototype</title>
  <script>
    Number.prototype.square =
      function () {
        return this.valueOf() * this.valueOf();
      }

    String.prototype.wrap =
      function (begin, end) {
        return begin + this.valueOf() + end;
      }

    console.log((13).square()); // 169
    console.log("Hi!".wrap("[", "]")); // [Hi!]
  </script>
```

```
</head>  
<body>  
  Listing 8-18: View the console output  
</body>  
</html>
```

 **NOTE:** It is not recommended to modify native object prototypes in a production environment, because this can often cause confusion by creating potential name collisions.

---

It is time to learn about a key topic, object inheritance, in the *next lesson*.