Modern JavaScript for the Impatient

Cay S. Horstmann

Modern JavaScript for the Impatient

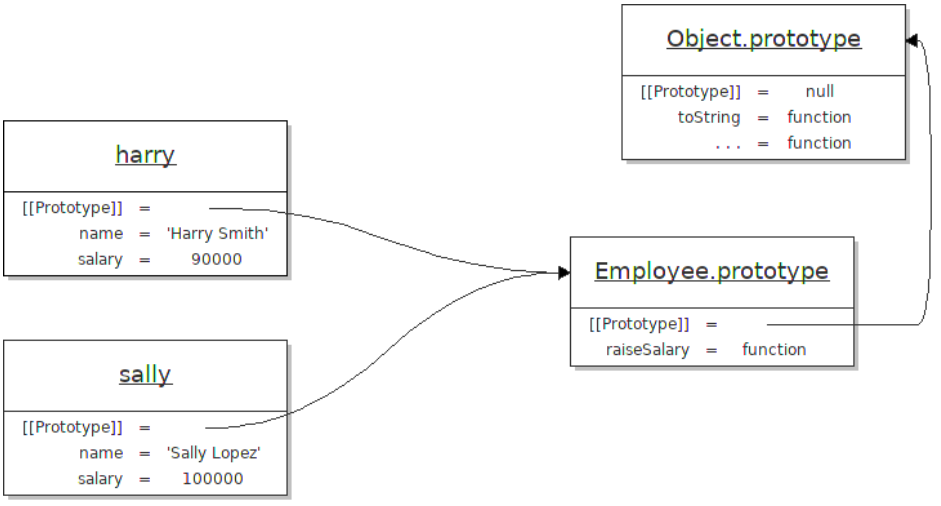

Figure 4-3 Objects created with a constructor

JavaScript, constructor functions are the equivalent of classes in class-based programming languages.

You won't often need to worry about the difference between traditional classes and the prototype-based system of JavaScript. As you will see in the following section, modern JavaScript syntax closely follows the conventions of class-based languages. However, every once in a while, you should remind yourself that a JavaScript class is nothing more than a constructor function, and that the common behavior is achieved with prototypes.

4.4 The Class Syntax

Nowadays, JavaScript has a class syntax that bundles up a constructor function and prototype methods in a familiar form. Here is the class syntax for the example of the preceding section:

```
class Employee {
  constructor(name, salary) {
    this.name = name
    this.salary = salary
  }
  raiseSalary(percent) {
    this.salary *= 1 + percent / 100
  }
}
```

This syntax does *exactly* the same as that of the preceding section. There still is no actual class. Behind the scenes, the `class` declaration merely declares a constructor function `Employee`. The `constructor` keyword declares the body of the `Employee` constructor function. The `raiseSalary` method is added to `Employee.prototype`.

As in the preceding section, you construct an object by calling the constructor function with the `new` operator:

```
const harry = new Employee('Harry Smith', 90000)
```

NOTE: As mentioned in the preceding sections, the `constructor` should not return any value. However, if it does, it is ignored, and the `new` expression still returns the newly created object.

You should definitely use the `class` syntax. (This is the golden rule #4 in the preface.) The syntax gets a number of fiddly details right that you do not want to manage manually. Just realize that a JavaScript `class` is syntactic sugar for a constructor function and a prototype object holding the methods.

NOTE: A class can have at most one `constructor`.

If you declare a class without a `constructor`, it automatically gets a constructor function with an empty body.

CAUTION: Unlike in an object literal, in a `class` declaration you do not use commas to separate the method declarations.

NOTE: Classes, unlike functions, are not hoisted. You need to declare a class before you can construct an instance.

NOTE: The body of a class is executed in strict mode.

4.5 Getters and Setters

A *getter* is a method with no parameters that is declared with the keyword `get`:

```
class Person {
  constructor(last, first) {
    this.last = last;
    this.first = first
  }
  get fullName() { return `${this.last}, ${this.first}` }
}
```

You call the getter without parentheses, as if you accessed a property value:

```
const harry = new Person('Smith', 'Harry')
const harrysName = harry.fullName // 'Smith, Harry'
```

The `harry` object does not have a `fullName` property, but the getter method is invoked. You can think of a getter as a dynamically computed property.

You can also provide a *setter*, a method with one parameter:

```
class Person {
  . . .
  set fullName(value) {
    const parts = value.split(/,\s*/)
    this.last = parts[0]
    this.first = parts[1]
  }
}
```

The setter is invoked when assigning to `fullName`:

```
harry.fullName = 'Smith, Harold'
```

When you provide getters and setters, users of your class have the illusion of using properties, but you control the property values and any attempts to modify them.

4.6 Instance Fields and Private Methods

You can dynamically set an object property in the constructor or any method by assigning to `this.`*propertyName*. These properties work the same way as instance fields in a class-based language.

```
class BankAccount {
  constructor() { this.balance = 0 }
  deposit(amount) { this.balance += amount }
  . . .
}
```

Three proposals for alternative notations are in stage 3 in early 2020. You can list the names and initial values of the fields in the class declaration, like this:

```
class BankAccount {
  balance = 0
  deposit(amount) { this.balance += amount }
  . . .
}
```

A field is *private* (that is, inaccessible outside the methods of the class) when its name starts with `#`:

```
class BankAccount {
  #balance = 0
  deposit(amount) { this.#balance += amount }
  . . .
}
```

A method is private if its name starts with a `#`.

4.7 Static Methods and Fields

In a `class` declaration, you can declare a method as `static`. Such a method does not operate on any object. It is a plain function that is a property of the class. Here is an example:

```
class BankAccount {
  . . .
  static percentOf(amount, rate) { return amount * rate / 100 }
  . . .
  addInterest(rate) {
    this.balance += BankAccount.percentOf(this.balance, rate)
  }
}
```

To call a static method, whether inside or outside the class, add the class name, as in the example above.

Behind the scenes, the static method is a property of the constructor. In the olden days, one had to do that by hand:

```
BankAccount.percentOf = function(amount, rate) {
  return amount * rate / 100
}
```

In the same way, you can define the equivalent of static fields:

```
BankAccount.OVERDRAFT_FEE = 30
```

In early 2020, a class-based syntax for static fields is in proposal stage 3:

```
class BankAccount {
  static OVERDRAFT_FEE = 30
  . . .
  withdraw(amount) {
    if (this.balance < amount) {
      this.balance -= BankAccount.OVERDRAFT_FEE
    }
    . . .
  }
}
```

A static field simply becomes a property of the constructor function. As with static methods, you access the field through the class name, as `BankAccount.OVERDRAFT_FEE`.

Private static fields and methods (prefixed with `#`) are also currently in proposal stage 3.

You can declare getters and setters as static methods. As always, the setter can do error checking:

```
class BankAccount {
  . . .
  static get OVERDRAFT_FEE() {
    return this.#OVERDRAFT_FEE // In a static method, this is the constructor function
  }
  static set OVERDRAFT_FEE(newValue) {
    if (newValue > this.#OVERDRAFT_FEE) {
      this.#OVERDRAFT_FEE = newValue
    }
  }
}
```

4.8 Subclasses

A key concept in object-oriented programming is inheritance. A class specifies behavior for its instances. You can form a subclass of a given class (called the superclass) whose instances behave differently in some respect, while inheriting other behavior from the superclass.

A standard teaching example is an inheritance hierarchy with a superclass `Employee` and a subclass `Manager`. While employees are expected to complete their assigned tasks in return for receiving their salary, managers get bonuses on top of their base salary if they actually achieve what they are supposed to do.

In JavaScript, as in Java, you use the `extends` keyword to express this relationship among the `Employee` and `Manager` classes:

```
class Employee {
  constructor(name, salary) { . . . }
  raiseSalary(percent) { . . . }
  . . .
}

class Manager extends Employee {
  getSalary() { return this.salary + this.bonus }
  . . .
}
```

Behind the scenes, a prototype chain is established—see Figure 4-4. The prototype of `Manager.prototype` is set to `Employee.prototype`. That way, any method that is not declared in the subclass is looked up in the superclass.

For example, you can call the `raiseSalary` on a manager object:

```
const boss = new Manager(. . .)
boss.raiseSalary(10) // Calls Employee.prototype.raiseSalary
```

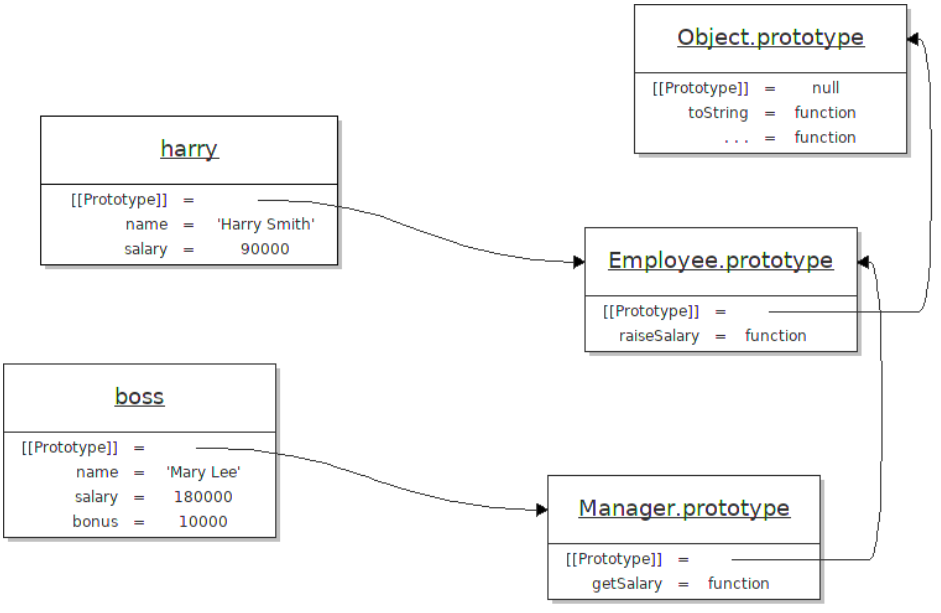

Figure 4-4 Prototype chain for inheritance

Prior to the `extends` syntax, JavaScript programmers had to establish such a prototype chain themselves.